BEA JRockit Mission Control®

BEA JRockit Runtime Analyzer

Mission Control version 3.0.1 ®
Document Revised: October, 2007

Copyright

Restricted Rights Legend

Trademarks and Service Marks

Contents

Introduction to the BEA JRockit Runtime Analyzer (JRA)

Workflow Description for Creating and Analyzing a JRA Recording

Alternative Ways to Start a JRA Recording

Using the JRA Tool

Getting Started with the BEA JRockit Runtime Analyzer Tool

General Information in a JRA Recording

Methods and Call Trace Information

General Garbage Collector Information

Garbage Collection Events Information

Java Heap Content Information

Objects Information

Code Optimization Information

Lock Profiling Information

Start and End Processes Information

Threads Information

Using the Latency Tabs

Latency Log Information

Latency Graph Information

Latency Traces Information

Adding Comments and Notes to a Recording

Introduction to the BEA JRockit Runtime Analyzer (JRA)

The BEA JRockit Runtime Analyzer (JRA) system is a Java application and JVM profiler that are especially designed for BEA JRockit. The JRA is a well integrated part of Mission Control and measures performance in a non-intrusive way in both production and development environments.

This section is divided into the following topics:

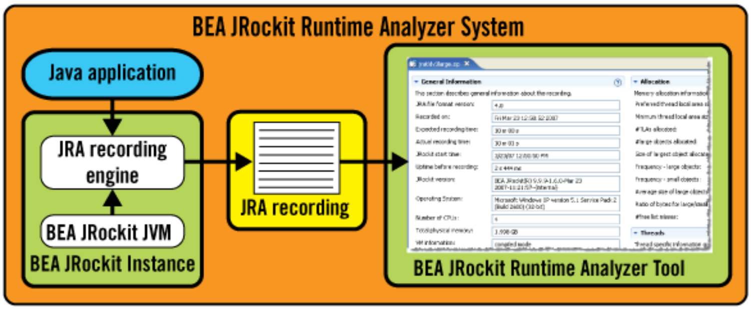
- How Does the JRockit Runtime Analyzer System Work?
- What is a JRA Recording?
- What is the JRA Tool?
- What's New in the JRA System?

How Does the JRockit Runtime Analyzer System Work?

The JRockit Runtime Analyzer (JRA) system consists of two parts (Figure 1-1): one part inside the JRockit JVM that collects and saves data (the JRA recording engine); and an analysis tool that visualizes the information (JRA Tool). 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 is automatically transferred to Mission Control and opened in the JRA Tool (this behavior is valid for JRockit 5.0 and later; for JRockit 1.4, the file is saved locally to disk and you need to locate it before opening the file).

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 JRockit 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), the JRockit thread analyzer (if enabled), and the JRockit lock profiler (if enabled).

What is a JRA Recording?

The JRA recording is a collection of data about the JVM and the running Java application. This recording can be used in the JRA Tool to analyze what happened in BEA JRockit and the Java application itself.

What is the JRA Tool?

The JRA Tool is a Java application that parses a 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.

What's New in the JRA System?

In Mission Control 3.0, the JRA has been extended to record even more information about your Java application and about the JVM itself. The JRA engine now has the possibility record thread related information and the JRA Tool has been extended with new tabs to visualize thread and thread latency information.

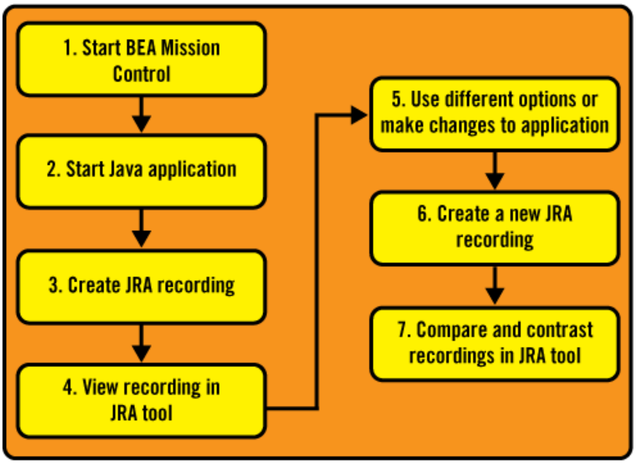
Another nifty feature that will make your JRA Tool workspace less cluttered, is that you can turn off tabs in the JRA Tool that are not showing any data. If you are a returning user to the JRA Tool, you will also find that the recording dialog for the JRA gives you more possibilities to take control over the recording itself, which data to include and not include, etc.

The thread and thread latency feature can help you pinpoint, down to the method, where you might have bottlenecks or problems in the application.

Workflow Description for Creating and Analyzing a JRA Recording

This section is a workflow description of how to use the JRA system to find problems and improvement areas with your Java application and BEA JRockit. JRA is excellent to use when tuning your system, for example when looking for performance bottlenecks, such as latencies. The typical workflow when working with the JRA is described in Figure 2-1.

Figure 2-1 Typical workflow for comparing different settings of JRockit and Java application


1. Start BEA Mission Control
2. Start Java application
3. Create JRA recording
4. View recording in JRA tool
5. Use different options or make changes to application
6. Create a new JRA recording
7. Compare and contrast recordings in JRA tool

The first steps are to start BEA Mission Control and then start you application so that you can start a JRA recording. The JRA recording takes a snapshot of the system's runtime behavior during the time period that you specify, typically a few minutes. As soon as the recording is complete, it opens in the JRA Tool where it can be analyzed "offline". If you want to, you can perform changes to your application or change command-line options for JRockit and create a new recording. This way, you have a chance to compare and contrast how different settings affect your application.

The steps for creating and comparing and contrasting a JRA recording are detailed in the following topics:

1. Start BEA JRockit Mission Control

The way you start Mission Control depends on which platform you are running it on.

Windows platforms:

Click **Start** > **All Programs** > **BEA JRockit** > **BEA JRockit Mission Control** or invoke the launcher (`JROCKIT_HOME\bin\jrmc.exe`).

Unix platforms:

```
JROCKIT_HOME/bin/jrmc
```

2. Start Your Java Application

1. Start your Java application with JRockit.
   - If you are running JRockit 1.5 and later and only want to monitor your application locally, you do not need to do anything else. It will be automatically discovered by JRockit Mission Control.
   - If you want to enable your application for remote monitoring, you need to add the `-Xmanagement` option to the command line. SSL and authentication will be enabled by default. If you do not wish to set up certificates ssl and authentication can be disabled by providing `ssl=false` and `authenticate=false`. Also, if you want to use the remote discovery feature of JRockit, you can enable it by setting

`autodiscovery=true`, for example,
`-Xmanagement:ssl=false,authenticate=false,autodiscovery=true`

2. Start your Java application and make sure it is running with a load. This way you get the best possible data collected for the JRA recording.

3. Create JRA Recording

It is simple to create a JRA recording, you only need to select a profile that you want to use. You can create recordings that either contain all "regular" JRA data or more extensive ones that also contain thread latency data. This section explains the difference between using the profile "JRA Recording Normal" without the advanced settings and with the advanced settings. There are alternative ways to start a JRA recording (see Alternative Ways to Start a JRA Recording).

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 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:`

The instructions for how to use the *JRA Recording Normal* profile is described in:

- To use the normal recording profile
- To use the normal recording profile with advanced options

For instructions on how to create a JRA recordings that includes latency data, see Creating a JRA Recording with Latency Data.

To use the normal recording profile

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

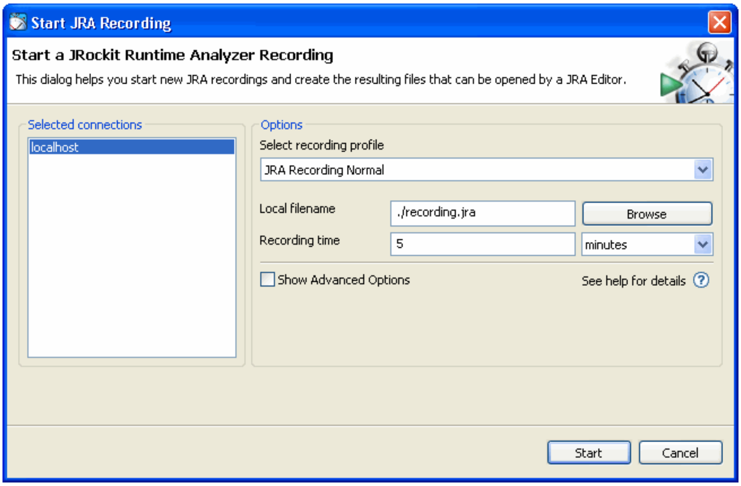
   If you run the application without load, the data captured from that application will not show where there is room for improvement.

2. In the **JRockit Browser**, select the JRockit instance you just started or select an entire folder with running JRockit instances.

3. Click the **Start JRA recording** button.

   The JRA Recording dialog box appears (Figure 2-2).

Figure 2-2 JRA Recording Dialog box



Start a JRockit Runtime Analyzer Recording

This dialog helps you start new JRA recordings and create the resulting files that can be opened by a JRA Editor.

Selected connections: localhost

Select recording profile: JRA Recording Normal

Local filename: ./recording.jra

Recording time: 5 minutes

☐ Show Advanced Options

4. Select the connection you want to record.
5. From the **Select recording file** drop-down list, choose **JRA Recording Normal**.

   This option is the "classic" JRA recording. This file contains all information that you as a returning user have found in JRA recordings for previous Mission Control and JRockit releases.

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 recording time for the duration of the recording in the **Recording time** field.
8. Select the time unit you wish to use for specifying the recording time (minutes or seconds).

   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.

9. Click **Start**.

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

To use the normal recording profile with advanced options

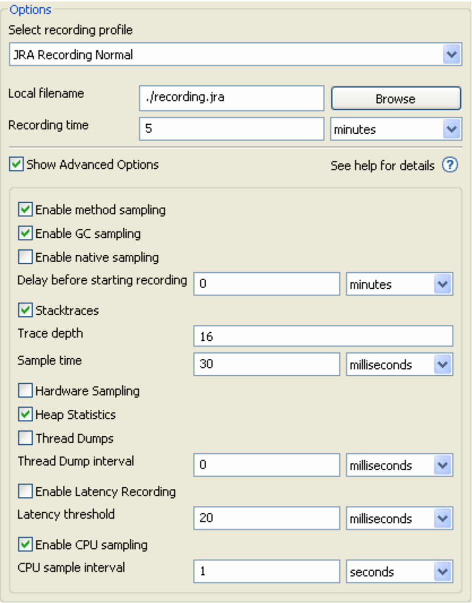
1. Make sure that your application is running and is under load.

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

2. In the **JRockit Browser**, select the JRockit instance you just started or select an entire folder with running JRockit instances (if you select a folder, you need to select the JRockit you want to monitor within the JRA Recording dialog box).
3. Select the connection you want to record.
4. From the **Select recording file** drop-down list, choose **JRA Recording Normal**.
5. Click **Show Advanced Options**.

   A panel with all options for creating a recording become visible (Figure 2-3).

Figure 2-3 JRA Recording with advanced option selected



Select recording profile: JRA Recording Normal
Local filename: ./recording.jra
Recording time: 5 minutes
☑ Show Advanced Options
☑ Enable method sampling
☑ Enable GC sampling
☐ Enable native sampling
Delay before starting recording: 0 minutes
☑ Stacktraces
Trace depth: 16
Sample time: 30 milliseconds
☐ Hardware Sampling
☑ Heap Statistics
☐ Thread Dumps
Thread Dump interval: 0 milliseconds
☐ Enable Latency Recording
Latency threshold: 20 milliseconds
☑ Enable CPU sampling
CPU sample interval: 1 seconds

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 recording time for the duration of the recording in the **Recording time** field.
8. Select the time unit you wish to use for specifying the recording time (minutes or seconds).

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.

9. Set a delay for when the recording should start in **Delay before starting recording**.

   It is good to set a delay if you know that your application has, for example, a long warm-up period.

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

   - **Enable method sampling**—records samples of methods.
   - **Enable GC sampling**—records garbage collection events.
   - **Enable native sampling**—records samples of native code.
   - **Stacktraces**—records method stack traces.
   - **Trace depth**—decides how “deep” (how many levels that they contain) the stack traces should go.
   - **Sample time**—decides how often samples should to be taken. If you set a small number, samples will be taken more frequently.
   - **Hardware sampling**—records sample data in the same manner as the command-line option `-xxhpm`.
   - **Heap statistics**—forces a garbage collection in the beginning and end of the recording to get Java heap data.
   - **Thread dumps**—creates thread dump data in the beginning and end of recording. If you set the Thread dump interval, you will also get thread dump data during the recording.
   - **Thread dump interval**—the time interval for how often thread dumps should be created. The thread dumps are displayed on the **Threads** tab in the JRA Tool.
   - **Enable Latency Recording**—creates latency data, see Creating a JRA Recording with Latency Data for more information.
   - **Latency threshold**—sets latency threshold, see Creating a JRA Recording with Latency Data for more information.
   - **Enable CPU sampling**—records JVM system and user load data and CPU usage. The information that is recorded is visible on the latency tabs.
   - **CPU sample interval**—sets the time interval for how often CPU sampling should be performed.

11. Click **Start**.

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

See Also

There are alternate ways to start a JRA recording, see Starting a Recording with jrcmd and Starting a Recording From the JRockit Command Line.

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. This can be fixed by turning off the option **Heap Statistics** in the JRA recording window (see Figure 2-3).

4. View Your JRA Recording in the JRA Tool

The recording results in an XML file that opens automatically in the JRA Tool upon completion. For JRockit 1.4 versions, the XML file is saved to the disk where JRockit is running and you need to open the JRA Tool prior to opening the JRA recording (see To open a JRA recording that was created with a JRockit 1.4 version).

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

There are several ways to open a JRA recording:

- To open a JRA recording by dragging and dropping
- To open a JRA recording within JRockit Mission Control
- To open a JRA recording that was created with a JRockit 1.4 version

To open a JRA recording 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 recording within 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 6-1).

   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”.

To open a JRA recording that was created with a JRockit 1.4 version

1. Start the JRA Tool with `java -jar RuntimeAnalyzer.jar`.
2. Click **File** > **Open file**.
3. Locate and select the recorded file and click **Open**.

   The Improve JRockit window opens. In this window you find information on how you can help the JRockit engineering team improving JRockit and the JRA.
4. Click **OK**.

   The **General** tab opens.

5. Perform Changes in Application or Use Other Command-line Options for BEA JRockit

For your second recording, you should make changes to either your Java application or the command-line options of BEA JRockit. Typical changes can be setting a different heap size on the nursery or changing the garbage collector in JRockit. Another good comparison could be to start your application with a newer version of JRockit, to see if the out-of-the box performance gives you better and more desired results.

6. Create a New JRA Recording

Create a new recording with the new settings or other JRockit version. The recording that you are comparing should be of the same length for optimal comparable data. See 3. Create JRA Recording for information on how to start a recording.

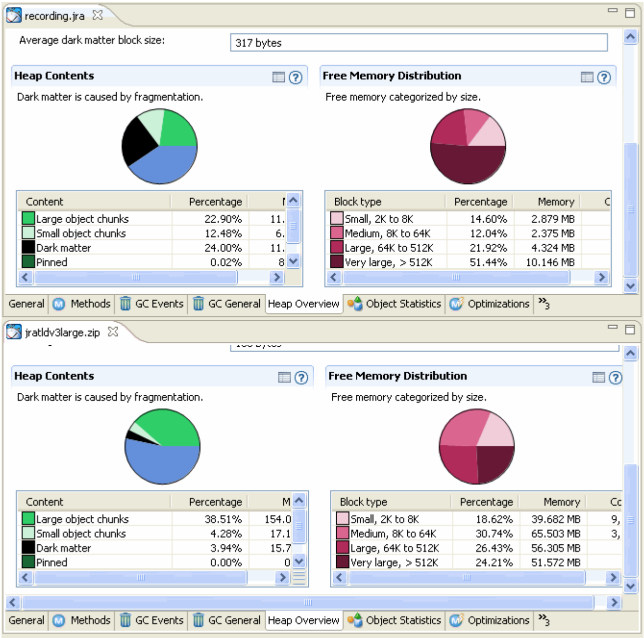
7. Compare and Contrast Two Recordings in the JRA Tool

The JRA Tool is excellent to use for comparing and contrasting recordings. Open both recordings in the JRA Tool and lay them next to each other to compare the results.

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 (Figure 2-4).

Figure 2-4 Comparing two JRA recordings in the JRA Tool



recording.jra

Average dark matter block size: 317 bytes

Heap Contents

Dark matter is caused by fragmentation.

Content	Percentage	I
🟩 Large object chunks	22.90%	11.
🟩 Small object chunks	12.48%	6.
■ Dark matter	24.00%	11.
🟩 Pinned	0.02%	8

Free Memory Distribution

Free memory categorized by size.

Block type	Percentage	Memory	C
🟥 Small, 2K to 8K	14.60%	2.879 MB	
🟥 Medium, 8K to 64K	12.04%	2.375 MB	
🟥 Large, 64K to 512K	21.92%	4.324 MB	
🟥 Very large, > 512K	51.44%	10.146 MB	

jratldv3large.zip

Heap Contents

Dark matter is caused by fragmentation.

Content	Percentage	M
🟩 Large object chunks	38.51%	154.0
🟩 Small object chunks	4.28%	17.1
■ Dark matter	3.94%	15.7
🟩 Pinned	0.00%	0

Free Memory Distribution

Free memory categorized by size.

Block type	Percentage	Memory	Cc
🟥 Small, 2K to 8K	18.62%	39.682 MB	9,
🟥 Medium, 8K to 64K	30.74%	65.503 MB	3,
🟥 Large, 64K to 512K	26.43%	56.305 MB	
🟥 Very large, > 512K	24.21%	51.572 MB	

Figure 2-4 shows the difference in Java heap content between two JRA recordings. The upper recording has much more dark matter than the lower one. The dark matter can cause disk fragmentation and will eventually slow down your application.

Alternative Ways to Start a JRA Recording

The default behavior is to start the JRA recording from within Mission Control (see 3. Create JRA Recording), but there are two alternate ways to start a recording. This section describes the two alternative ways to start a JRA recording.

- Starting a Recording with jrcmd
- Starting a Recording From the JRockit Command Line

Starting 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.

Table 3-1 Command Line Startup Options

Option	Description
`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.
`heapstat=<true | false>`	Allows you to enable or disable the tracking of heap statistics. • `-XXjra:heapstat=true` enables heap statistic tracking • `-XXjra:heapstat=false` disables heap statistic tracking. This tracking is enabled by default but, under certain circumstances can adversely affect transaction latency. In those situations, it is strongly recommended that you disable heap statistic tracking.

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.

The startup options that you have used are shown in the VM Arguments tab on the **General** tab. See View VM Arguments.

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`).

CHAPTER 4

Using the JRA Tool

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

- Getting Started with the BEA JRockit Runtime Analyzer Tool
- General Information in a JRA Recording
- Methods and Call Trace Information
- Garbage Collection Events Information
- General Garbage Collector Information
- Java Heap Content Information
- Objects Information
- Code Optimization Information
- Lock Profiling Information
- Start and End Processes Information
- Threads Information
- Using the Latency Tabs
- Latency Log Information
- Latency Graph Information
- Latency Traces Information

- Adding Comments and Notes to a Recording

CHAPTER 5

Getting Started with the BEA JRockit Runtime Analyzer Tool

A JRA recording comes with a wealth of information that might seem cumbersome to interpret at first. You need to keep in mind, however, that the recording should be used when you know that you have a problem with your application, then the JRA information can help you visualize those problems so that you have a better chance of fixing them.

This topic gives an overview of the JRA Tool components and how to customize the tool itself. It includes the following sections:

- Starting the JRA Tool
- JRA Tool Overview
- Customizing Your JRA Tool

Starting the JRA Tool

There are two ways the JRA Tool is started: either automatically when you have created a recording (see To use the normal recording profile) or when you open an already existing recording (see To open a JRA recording within JRockit Mission Control).

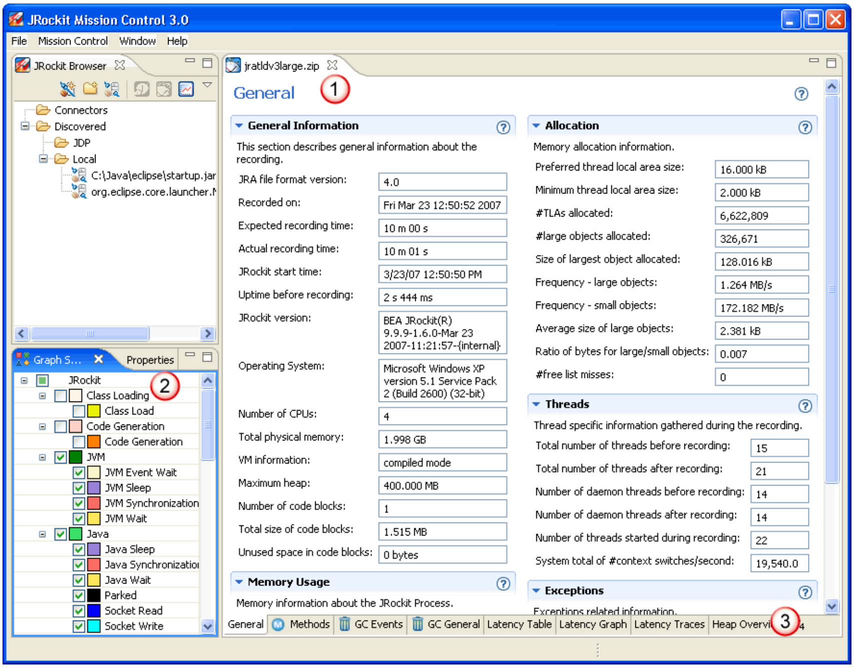
Note: If you are running a JRockit based on Java 1.4, the JRA Tool does not open automatically when the recording is completed.

JRA Tool Overview

The JRA Tool is a multi-tabbed interface, each tab allowing you to monitor different aspects of a JRA recording. New for Mission Control 3.0, is that you can view thread information and thread latency information. When all types of recording data has been collected and when all tabs are activated, the JRA Tool includes eleven tabs within the main window (Figure 5-1). When you view Latency information, the extra tabs (outside the main JRA Tool window) Event Types and Properties are also used (marked 2 in Figure 5-1).

Note: The number of tabs that are displayed depends on the JRA recording itself (if all sample data has been collected or not) and settings in the Properties window (see Turning on/off Tabs).

Figure 5-1 JRA Tool overview



General Information

This section describes general information about the recording.

JRA file format version: 4.0
Recorded on: Fri Mar 23 12:50:52 2007
Expected recording time: 10 m 00 s
Actual recording time: 10 m 01 s
JRockit start time: 3/23/07 12:50:50 PM
Uptime before recording: 2 s 444 ms
JRockit version: BEA JRockit(R) 9.9.9-1.6.0-Mar 23 2007-11:21:57-{internal}
Operating System: Microsoft Windows XP version 5.1 Service Pack 2 (Build 2600) (32-bit)
Number of CPUs: 4
Total physical memory: 1.998 GB
VM information: compiled mode
Maximum heap: 400.000 MB
Number of code blocks: 1
Total size of code blocks: 1.515 MB
Unused space in code blocks: 0 bytes

Memory Usage

Memory information about the JRockit Process.

Allocation

Memory allocation information.

Preferred thread local area size: 16.000 kB
Minimum thread local area size: 2.000 kB
#TLAs allocated: 6,622,809
#large objects allocated: 326,671
Size of largest object allocated: 128.016 kB
Frequency - large objects: 1.264 MB/s
Frequency - small objects: 172.182 MB/s
Average size of large objects: 2.381 kB
Ratio of bytes for large/small objects: 0.007
#free list misses: 0

Threads

Thread specific information gathered during the recording.

Total number of threads before recording: 15
Total number of threads after recording: 21
Number of daemon threads before recording: 14
Number of daemon threads after recording: 14
Number of threads started during recording: 22
System total of #context switches/second: 19,540.0

Exceptions

Exceptions related information.

The main JRA Tool window is divided into the following sections:

1. The main JRA Tool window—the available tabs depends on settings in the Preferences window and the type of data collected in the JRA recording.
2. Tabs that are valid for Latency trouble shooting only.
3. Tabs for different aspects of the JRA recording.

JRA Tabs at a Glance

The following information about the tabs are available:

- Getting Familiar with the General Tab
- Getting Familiar with the Methods Tab
- Getting Familiar with the GC General Tab
- Getting Familiar with the GCs Tab
- Getting Familiar with the Heap Tab
- Getting Familiar with the Objects Tab
- Getting Familiar with the Optimizations Tab
- Getting Familiar with the Locks Tab
- Getting Familiar with the Processes Tab
- Getting Familiar with the Threads Tab
- Getting Familiar with the Latency Log Tab
- Getting Familiar with the Latency Graph Tab
- Getting Familiar with the Latency Traces Tab
- Adding Comments and Notes to a Recording

Customizing Your JRA Tool

You can customize your JRA Tool in the following ways:

- Turning on/off Tabs
- Changing Table Settings
- Filtering Information
- Collapsing and Expanding an Information Panel
- Changing Layout of a Tab

Turning on/off Tabs

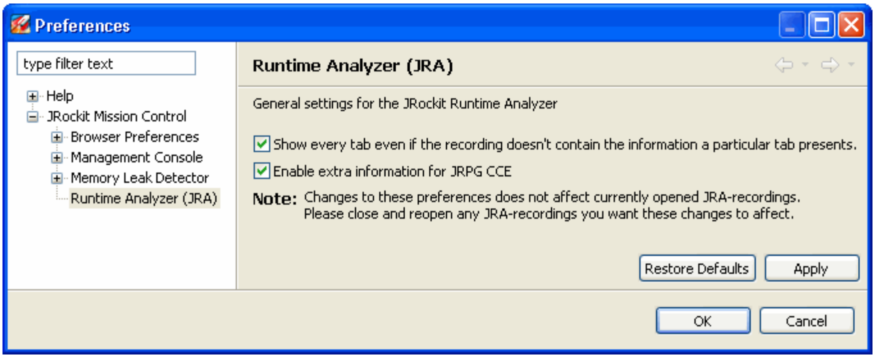
When you create a JRA recording, there are several options that you can choose to record or not (see 3. Create JRA Recording). If you decide to exclude something from the recording, the JRA Tool automatically excludes the tab that does not contain any information. This way, you will not get so many tabs to maneuver within the JRA Tool. You can, however, have the JRA Tool show all tabs, by turning on that function in the Preferences window.

To set preferences for the JRA Tool

1. Click **Window** > **Preferences**.

   The **Preferences** window opens (Figure 5-2).

Figure 5-2 Setting preferences in the JRA Tool



Runtime Analyzer (JRA)

General settings for the JRockit Runtime Analyzer

☑ Show every tab even if the recording doesn't contain the information a particular tab presents.

☑ Enable extra information for JRPG CCE

Note: Changes to these preferences does not affect currently opened JRA-recordings. Please close and reopen any JRA-recordings you want these changes to affect.

2. Select none, one, or both of the JRA preferences:
   - **Show every tab**...—when you choose to see every tab, the JRA Tool shows all tabs in the interface regardless of if the tab contains any information.
   - **Enable extra information**...—the extra information is only useful to BEA JRockit support personnel and this option is used if you have been asked to send a JRA recording to your BEA support representative.
3. Click **Apply** for the settings to take effect.
4. Click **OK** to close the **Preferences** window.

Changing Table Settings

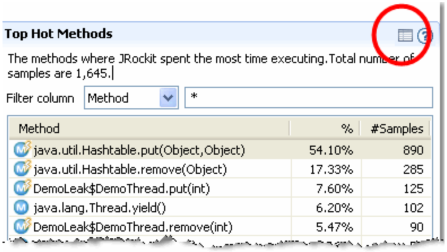
The JRA Tool lists a lot of information in different tables. These tables can be customized to display information of your choice. You can also preset the width of the columns in the tables.

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 (Figure 5-3).

Figure 5-3 Table settings button



Top Hot Methods

The methods where JRockit spent the most time executing. Total number of samples are 1,645.

Filter column: Method *

Method	%	#Samples
java.util.Hashtable.put(Object,Object)	54.10%	890
java.util.Hashtable.remove(Object)	17.33%	285
DemoLeak$DemoThread.put(int)	7.60%	125
java.lang.Thread.yield()	6.20%	102
DemoLeak$DemoThread.remove(int)	5.47%	90

A **Table settings** window appears (Figure 5-4).

Figure 5-4 Table settings window

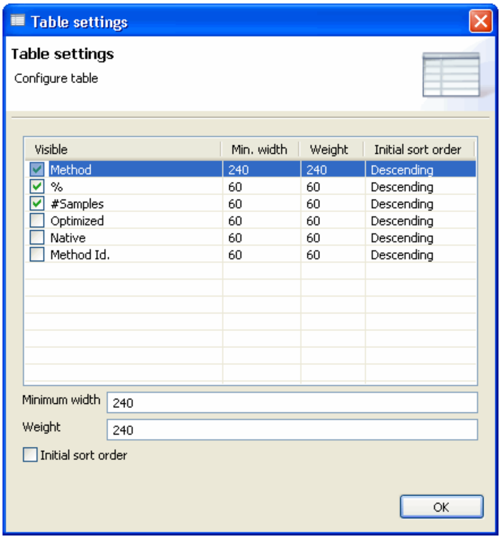


Table settings

Configure table

Visible	Min. width	Weight	Initial sort order
☑ Method	240	240	Descending
☑ %	60	60	Descending
☑ #Samples	60	60	Descending
☐ Optimized	60	60	Descending
☐ Native	60	60	Descending
☐ Method Id.	60	60	Descending

Minimum width 240

Weight 240

☐ Initial sort order

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**.

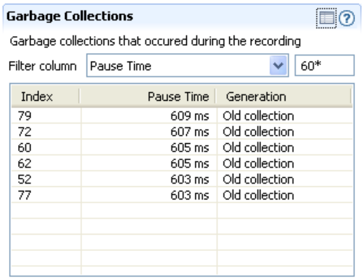
Filtering Information

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 5-5, **Pause Time** was selected.
2. Enter a number or text for the information you want to see. In this example, Figure 5-5, ***60**** was used to see all Pause Times that contains a value starting with 60.

Figure 5-5 Filtering information



Garbage Collections

Garbage collections that occured during the recording

Filter column: Pause Time — 60*

Index	Pause Time	Generation
79	609 ms	Old collection
72	607 ms	Old collection
60	605 ms	Old collection
62	605 ms	Old collection
52	603 ms	Old collection
77	603 ms	Old collection

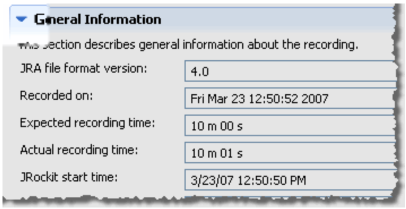
Collapsing and Expanding an Information Panel

Sometimes the information on a tab can be cumbersome to work with, then it is good to collapse the view of the panels that you are not working with or viewing.

To collapse/expand a view

- Click on the small arrow next to a description field (see highlight in Figure 5-6) to collapse the view of the General Information field.

Figure 5-6 Collapsing a view



General Information

section describes general information about the recording.

JRA file format version: 4.0
Recorded on: Fri Mar 23 12:50:52 2007
Expected recording time: 10 m 00 s
Actual recording time: 10 m 01 s
JRockit start time: 3/23/07 12:50:50 PM

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 Layout of a Tab

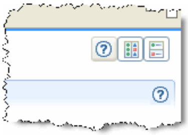
Sometimes the method names are hard to view in the default horizontal layout, therefore, you might want to change the layout to a vertical view instead.

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 (Figure 5-7).

Note: Not all tabs have this functionality.

Figure 5-7 Horizontal and Vertical layout buttons



CHAPTER 6

General Information in a 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. That general information is displayed on the **General tab** in the JRA Tool.

For recordings that have been generated with a JRockit that is older than R26.4, you should still be able to open them 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 newer releases.

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

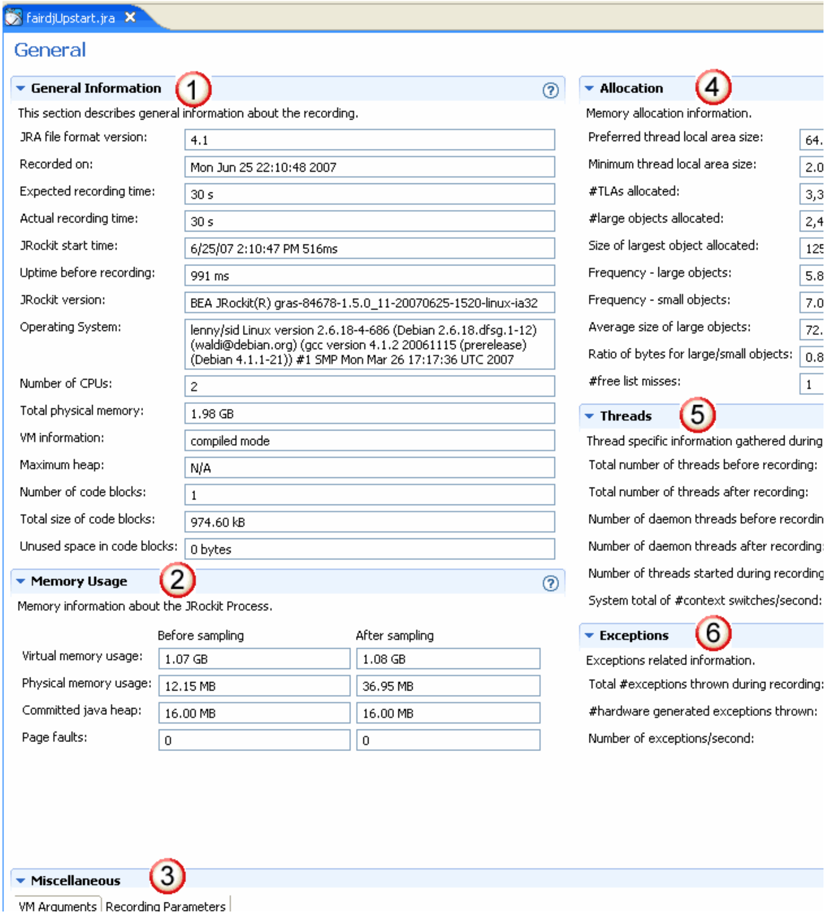
This section is divided into the following topics:

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

Getting Familiar with the General Tab

The **General** tab (Figure 6-1) contains information on both JRockit, your system, and your application.

Figure 6-1 The General tab



fairdjUpstart.jra

General

General Information

This section describes general information about the recording.

JRA file format version: 4.1
Recorded on: Mon Jun 25 22:10:48 2007
Expected recording time: 30 s
Actual recording time: 30 s
JRockit start time: 6/25/07 2:10:47 PM 516ms
Uptime before recording: 991 ms
JRockit version: BEA JRockit(R) gras-84678-1.5.0_11-20070625-1520-linux-ia32
Operating System: lenny/sid Linux version 2.6.18-4-686 (Debian 2.6.18.dfsg.1-12) (waldi@debian.org) (gcc version 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)) #1 SMP Mon Mar 26 17:17:36 UTC 2007
Number of CPUs: 2
Total physical memory: 1.98 GB
VM information: compiled mode
Maximum heap: N/A
Number of code blocks: 1
Total size of code blocks: 974.60 kB
Unused space in code blocks: 0 bytes

Memory Usage

Memory information about the JRockit Process.

	Before sampling	After sampling
Virtual memory usage:	1.07 GB	1.08 GB
Physical memory usage:	12.15 MB	36.95 MB
Committed java heap:	16.00 MB	16.00 MB
Page faults:	0	0

Allocation

Memory allocation information.

Preferred thread local area size: 64.
Minimum thread local area size: 2.0
#TLAs allocated: 3,3
#large objects allocated: 2,4
Size of largest object allocated: 12
Frequency - large objects: 5.8
Frequency - small objects: 7.0
Average size of large objects: 72.
Ratio of bytes for large/small objects: 0.8
#free list misses: 1

Threads

Thread specific information gathered during
Total number of threads before recording:
Total number of threads after recording:
Number of daemon threads before recordin
Number of daemon threads after recording
Number of threads started during recording
System total of #context switches/second:

Exceptions

Exceptions related information.
Total #exceptions thrown during recording:
#hardware generated exceptions thrown:
Number of exceptions/second:

Miscellaneous

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.

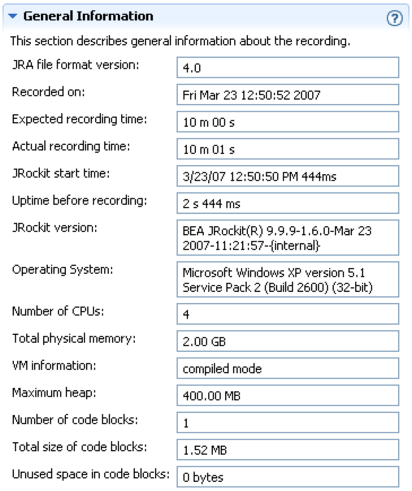
3. **Miscellaneous**—contains additionalinformation about a recording. This section is divided into two tabbed panels:
   - **VM Arguments**—lists all startup options that were used.
   - **Recording Parameters**—lists all the configurable options used during the recording and the values seet for them.
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 panel displays (Figure 6-2) 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 6-2 General Information panel



General Information

This section describes general information about the recording.

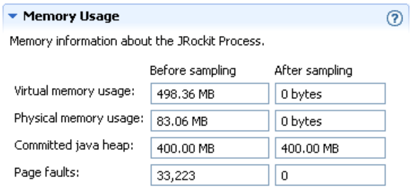
JRA file format version:	4.0
Recorded on:	Fri Mar 23 12:50:52 2007
Expected recording time:	10 m 00 s
Actual recording time:	10 m 01 s
JRockit start time:	3/23/07 12:50:50 PM 444ms
Uptime before recording:	2 s 444 ms
JRockit version:	BEA JRockit(R) 9.9.9-1.6.0-Mar 23 2007-11:21:57-{internal}
Operating System:	Microsoft Windows XP version 5.1 Service Pack 2 (Build 2600) (32-bit)
Number of CPUs:	4
Total physical memory:	2.00 GB
VM information:	compiled mode
Maximum heap:	400.00 MB
Number of code blocks:	1
Total size of code blocks:	1.52 MB
Unused space in code blocks:	0 bytes

Viewing Memory Usage Information

This panel (Figure 6-3) 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 6-3 Memory Usage panel



Memory Usage

Memory information about the JRockit Process.

	Before sampling	After sampling
Virtual memory usage:	498.36 MB	0 bytes
Physical memory usage:	83.06 MB	0 bytes
Committed java heap:	400.00 MB	400.00 MB
Page faults:	33,223	0

Viewing Miscellaneous Information

The Miscellaneous panel is a tabbed interface that shows information that can help you better understand a recording. In this section, you can:

- View VM Arguments
- View Recording Parameters

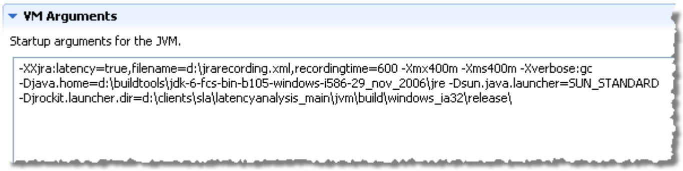
View VM Arguments

This panel displays (Figure 6-4) 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 records latencies (`XXjra:latency`) has been set (100 seconds).
- The name of the recorded file has been set (`filename`) and the duration of the recording (`recordingtime`).
- The initial, minimum and maximum Java heap has been set (`-Xms` and `-Xmx`)
- Some non-standard (BEA internal) D-options have also been set in this example.

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 6-4 VM Arguments



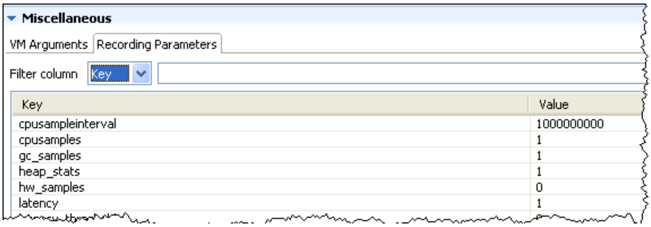
Startup arguments for the JVM.

```
-XXjra:latency=true,filename=d:\jrarecording.xml,recordingtime=600 -Xmx400m -Xms400m -Xverbose:gc
-Djava.home=d:\buildtools\jdk-6-fcs-bin-b105-windows-i586-29_nov_2006\jre -Dsun.java.launcher=SUN_STANDARD
-Djrockit.launcher.dir=d:\clients\sla\latencyanalysis_main\jvm\build\windows_ia32\release\
```

View Recording Parameters

This panel displays all configurable options used in the current recording and the values attributed to those options.

Figure 6-5 Recording Parameters



Filter column: Key

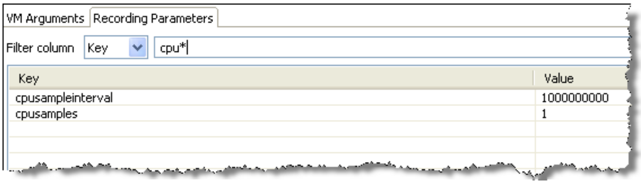
Key	Value
cpusampleinterval	1000000000
cpusamples	1
gc_samples	1
heap_stats	1
hw_samples	0
latency	1

You can determine which options you can see by specifying a filter. To filter options, do the following

1. Select a table column name for which you want to filter the information. In Figure 6-6, the **Key** column was selected.

2. Enter text for the information you want to see; for example, in Figure 6-6, `cpu*` was entered to limit the options displayed to just those beginning with the text string “cpu”.

Figure 6-6 Filtered Recording Paramters



Filter column: Key — cpu*

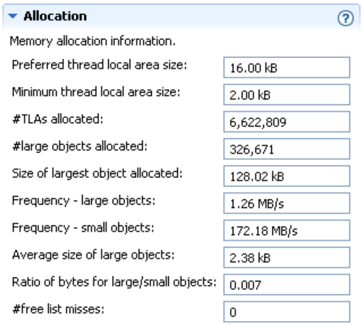
Key	Value
cpusampleinterval	1000000000
cpusamples	1

Viewing Memory Allocation Information

This panel displays information about how JRockit is allocating memory on the Java heap (Figure 6-7). A **Thread Local Area (TLA)** 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. For an in-depth explanation of how TLA works, please see Setting the Thread Local Area Size in the BEA JRockit Diagnostics Guide. See also `-XXtlaSize` in the BEA JRockit Reference Manual for more information on how to set different values of the TLA size.

- **Preferred thread local area (TLA) size** is the value that you have set with the command-line option `-XXtlasize:preferred <size>`.
- **Minimum thread local area size** is the value that you have set with the command-line option `-XXtlasize:min <size>`.
- **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 6-7 Allocation panel



Allocation

Memory allocation information.

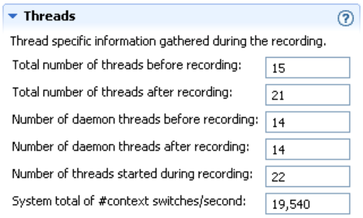
Preferred thread local area size:	16.00 kB
Minimum thread local area size:	2.00 kB
#TLAs allocated:	6,622,809
#large objects allocated:	326,671
Size of largest object allocated:	128.02 kB
Frequency - large objects:	1.26 MB/s
Frequency - small objects:	172.18 MB/s
Average size of large objects:	2.38 kB
Ratio of bytes for large/small objects:	0.007
#free list misses:	0

Viewing Threads Information

This panel displays (Figure 6-8) information on the number of Java threads that existed both before and after the recording.

- The value of **Total number of threads before/after recording** shows how many threads were active before the recording started and how many were active when the recording ended.
- 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 **Number of threads started during recording** shows how many threads were started.
- 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 6-8 Threads panel



Threads

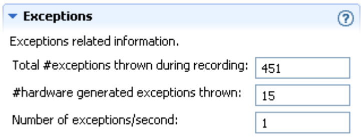
Thread specific information gathered during the recording.

Total number of threads before recording:	15
Total number of threads after recording:	21
Number of daemon threads before recording:	14
Number of daemon threads after recording:	14
Number of threads started during recording:	22
System total of #context switches/second:	19,540

Viewing Exceptions Information

This panel displays (Figure 6-9) 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 6-9 Exceptions information



Exceptions related information.

Total #exceptions thrown during recording: 451

#hardware generated exceptions thrown: 15

Number of exceptions/second: 1

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.

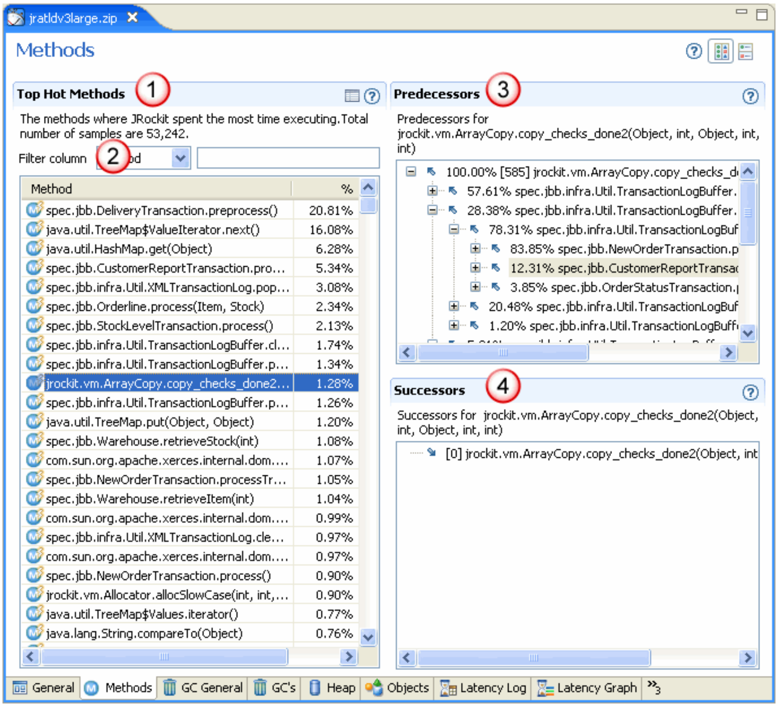
This section is divided into the following topics:

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

Getting Familiar with the Methods Tab

The **Methods tab** (Figure 7-1) lists the top hot methods, with its predecessors and successors, that were recorded.

Figure 7-1 The Methods tab



The methods where JRockit spent the most time executing.Total number of samples are 53,242.

Method	%
spec.jbb.DeliveryTransaction.preprocess()	20.81%
java.util.TreeMap$ValueIterator.next()	16.08%
java.util.HashMap.get(Object)	6.28%
spec.jbb.CustomerReportTransaction.pro...	5.34%
spec.jbb.infra.Util.XMLTransactionLog.pop...	3.08%
spec.jbb.Orderline.process(Item, Stock)	2.34%
spec.jbb.StockLevelTransaction.process()	2.13%
spec.jbb.infra.Util.TransactionLogBuffer.cl...	1.74%
spec.jbb.infra.Util.TransactionLogBuffer.p...	1.34%
jrockit.vm.ArrayCopy.copy_checks_done2...	1.28%
spec.jbb.infra.Util.TransactionLogBuffer.p...	1.26%
java.util.TreeMap.put(Object, Object)	1.20%
spec.jbb.Warehouse.retrieveStock(int)	1.08%
com.sun.org.apache.xerces.internal.dom....	1.07%
spec.jbb.NewOrderTransaction.processTr...	1.05%
spec.jbb.Warehouse.retrieveItem(int)	1.04%
com.sun.org.apache.xerces.internal.dom....	0.99%
spec.jbb.infra.Util.XMLTransactionLog.cle...	0.97%
com.sun.org.apache.xerces.internal.dom....	0.97%
spec.jbb.NewOrderTransaction.process()	0.90%
jrockit.vm.Allocator.allocSlowCase(int, int,...	0.90%
java.util.TreeMap$Values.iterator()	0.77%
java.lang.String.compareTo(Object)	0.76%

Predecessors for jrockit.vm.ArrayCopy.copy_checks_done2(Object, int, Object, int, int)

```
⊟ 100.00% [585] jrockit.vm.ArrayCopy.copy_checks_d
    ⊞ 57.61% spec.jbb.infra.Util.TransactionLogBuffer.
    ⊟ 28.38% spec.jbb.infra.Util.TransactionLogBuffer.
        ⊟ 78.31% spec.jbb.infra.Util.TransactionLogBuf
            ⊞ 83.85% spec.jbb.NewOrderTransaction.p
            ⊞ 12.31% spec.jbb.CustomerReportTransac
            ⊞ 3.85% spec.jbb.OrderStatusTransaction.
        ⊞ 20.48% spec.jbb.infra.Util.TransactionLogBuf
        ⊞ 1.20% spec.jbb.infra.Util.TransactionLogBuff
```

Successors for jrockit.vm.ArrayCopy.copy_checks_done2(Object, int, Object, int, int)

```
[0] jrockit.vm.ArrayCopy.copy_checks_done2(Object, int
```

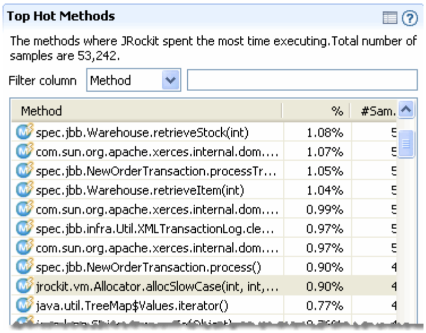
The **Methods** tab is divided into the following sections:

1. **Top Hot Methods**—a listing of the top hot methods. Click on the different table headings to get a different sort order.
2. **Filter column**—see Filtering Information on how to use this function.
3. **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 panel.
4. **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 panel.

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** table (Figure 7-2) lists 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 7-2 Top Hot Methods shown



Top Hot Methods

The methods where JRockit spent the most time executing.Total number of samples are 53,242.

Filter column: Method

Method	%	#Sam.
spec.jbb.Warehouse.retrieveStock(int)	1.08%	5
com.sun.org.apache.xerces.internal.dom....	1.07%	5
spec.jbb.NewOrderTransaction.processTr...	1.05%	5
spec.jbb.Warehouse.retrieveItem(int)	1.04%	5
com.sun.org.apache.xerces.internal.dom....	0.99%	5
spec.jbb.infra.Util.XMLTransactionLog.cle...	0.97%	5
com.sun.org.apache.xerces.internal.dom....	0.97%	5
spec.jbb.NewOrderTransaction.process()	0.90%	4
jrockit.vm.Allocator.allocSlowCase(int, int,...	0.90%	4
java.util.TreeMap$Values.iterator()	0.77%	4

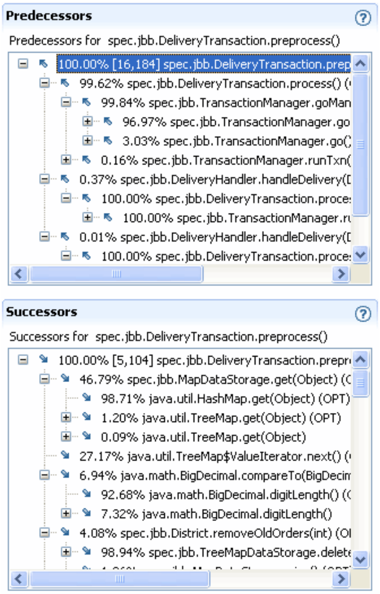
Note: If your recording has native sampling enabled during the recording, you can see methods prefixed by `jvm`, which are native methods in JRockit.

Use the filtering function to find the method you are looking for, see Filtering Information.

Viewing Predecessors and Successors

By selecting a method in the **Top Hot Methods** table, you can see its sampled **Predecessors** and **Successors** (Figure 7-3). The predecessors are the methods that call the selected method and the successors are the methods that the selected method calls.

Figure 7-3 Viewing Predecessors and Successors



```
Predecessors for  spec.jbb.DeliveryTransaction.preprocess()
⊟ 100.00% [16,184] spec.jbb.DeliveryTransaction.prep
    ⊟ 99.62% spec.jbb.DeliveryTransaction.process() (
        ⊟ 99.84% spec.jbb.TransactionManager.goMan
            ⊞ 96.97% spec.jbb.TransactionManager.go
            ⊞ 3.03% spec.jbb.TransactionManager.go(
        ⊞ 0.16% spec.jbb.TransactionManager.runTxn(
    ⊟ 0.37% spec.jbb.DeliveryHandler.handleDelivery(
        ⊟ 100.00% spec.jbb.DeliveryTransaction.proce
            ⊞ 100.00% spec.jbb.TransactionManager.r
    ⊟ 0.01% spec.jbb.DeliveryHandler.handleDelivery(
        ⊟ 100.00% spec.jbb.DeliveryTransaction.proce
```

```
Successors for  spec.jbb.DeliveryTransaction.preprocess()
⊟ 100.00% [5,104] spec.jbb.DeliveryTransaction.prepr
    ⊟ 46.79% spec.jbb.MapDataStorage.get(Object) (
          98.71% java.util.HashMap.get(Object) (OPT)
        ⊞ 1.20% java.util.TreeMap.get(Object) (OPT)
        ⊞ 0.09% java.util.TreeMap.get(Object)
      27.17% java.util.TreeMap$ValueIterator.next() (
    ⊟ 6.94% java.math.BigDecimal.compareTo(BigDeci
          92.68% java.math.BigDecimal.digitLength() (
        ⊞ 7.32% java.math.BigDecimal.digitLength()
    ⊟ 4.08% spec.jbb.District.removeOldOrders(int) (O
        ⊞ 98.94% spec.jbb.TreeMapDataStorage.delet
```

The number within brackets of a particular predecessor or successor 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.

CHAPTER 8

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.

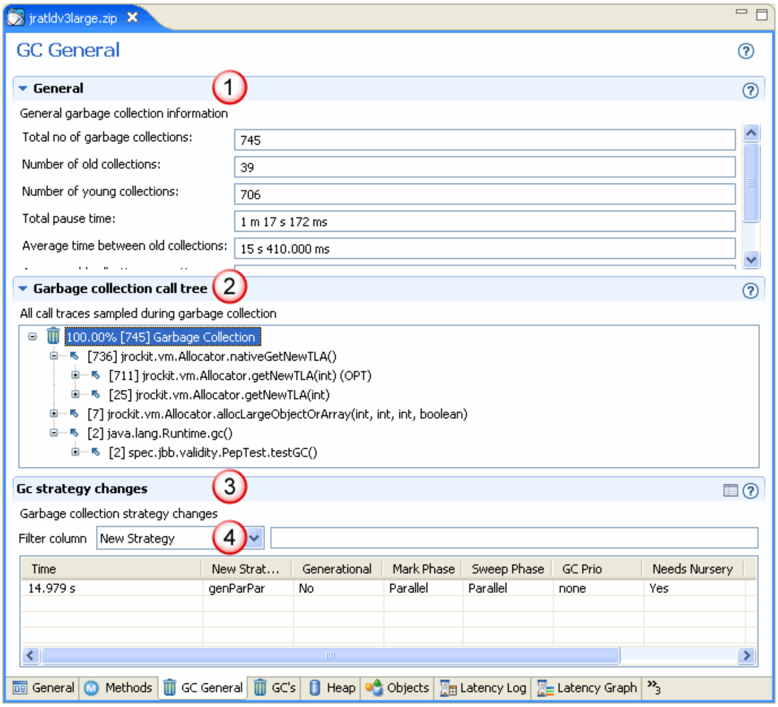
This section is divided into the following topics:

- Getting Familiar with the GC General Tab
- Viewing General Garbage Collection Information
- Viewing Garbage Collection Call Tree Information
- Viewing Garbage Collection Strategy Changes Information

Getting Familiar with the GC General Tab

The **GC General** tab (Figure 8-1) shows general information about a garbage collection, its call tree, and what garbage collection strategies have taken place.

Figure 8-1 The GC General tab



General garbage collection information

Total no of garbage collections: 745
Number of old collections: 39
Number of young collections: 706
Total pause time: 1 m 17 s 172 ms
Average time between old collections: 15 s 410.000 ms

All call traces sampled during garbage collection

```
100.00% [745] Garbage Collection
    [736] jrockit.vm.Allocator.nativeGetNewTLA()
        [711] jrockit.vm.Allocator.getNewTLA(int) (OPT)
        [25] jrockit.vm.Allocator.getNewTLA(int)
    [7] jrockit.vm.Allocator.allocLargeObjectOrArray(int, int, int, boolean)
    [2] java.lang.Runtime.gc()
        [2] spec.jbb.validity.PepTest.testGC()
```

Garbage collection strategy changes

Filter column: New Strategy

Time	New Strat...	Generational	Mark Phase	Sweep Phase	GC Prio	Needs Nursery
14.979 s	genParPar	No	Parallel	Parallel	none	Yes

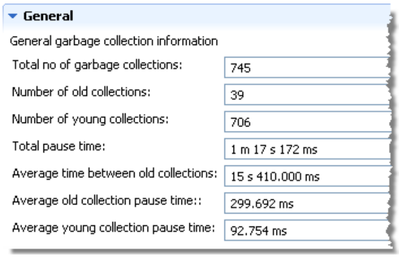
The **GC General** tab is divided into the following sections:

1. **General**—this panel shows overall statistics about the garbage collections during the entire JRA recording.
2. **Garbage Collection Call Tree**—this panel is a collection of all call traces that were sampled for all garbage collections for the JRA recording.
3. **GC Strategy Changes**—this table lists when a garbage collection strategy took place and how it changed.
4. **Filter column**—see Filtering Information on how to use this function.

Viewing General Garbage Collection Information

The **General** panel (Figure 8-2) shows general garbage collection information such as the total number of garbage collections during the recording and the duration of all pause times due to garbage collection. You can use this information to, for example, see whether your application is coming down to desired pause time averages or not.

Figure 8-2 General Garbage Collection Information



General garbage collection information

Total no of garbage collections: 745

Number of old collections: 39

Number of young collections: 706

Total pause time: 1 m 17 s 172 ms

Average time between old collections: 15 s 410.000 ms

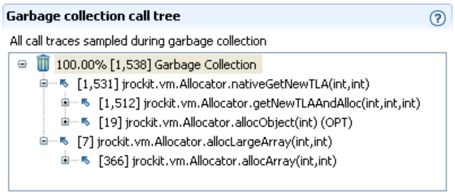
Average old collection pause time:: 299.692 ms

Average young collection pause time: 92.754 ms

Viewing Garbage Collection Call Tree Information

The **Garbage Collection Call Tree** panel (Figure 8-3) 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 8-3 Garbage Collection Call Tree Information



All call traces sampled during garbage collection

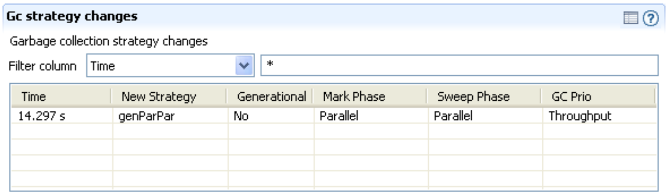
```
100.00% [1,538] Garbage Collection
    [1,531] jrockit.vm.Allocator.nativeGetNewTLA(int,int)
        [1,512] jrockit.vm.Allocator.getNewTLAAndAlloc(int,int,int)
        [19] jrockit.vm.Allocator.allocObject(int) (OPT)
    [7] jrockit.vm.Allocator.allocLargeArray(int,int)
        [366] jrockit.vm.Allocator.allocArray(int,int)
```

Viewing Garbage Collection Strategy Changes Information

The **Garbage Collection Strategy Changes** table (Figure 8-4) lists when the garbage collector has changed strategy, for example, JRockit has been set to run for best throughput (`-Xgcprio:throughput`, **GC Prio** in Figure 8-4), then JRockit changes strategy in runtime to best reach this goal. 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 8-4 Garbage Collection Strategy Changes Information



Gc strategy changes

Garbage collection strategy changes

Filter column: Time — *

Time	New Strategy	Generational	Mark Phase	Sweep Phase	GC Prio
14.297 s	genParPar	No	Parallel	Parallel	Throughput

In the example seen in Figure 8-4, there has been one strategy change for the garbage collector.

Use the filtering function to find a specific garbage collection, see Filtering Information.

Garbage Collection Events Information

The **GCs** 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.

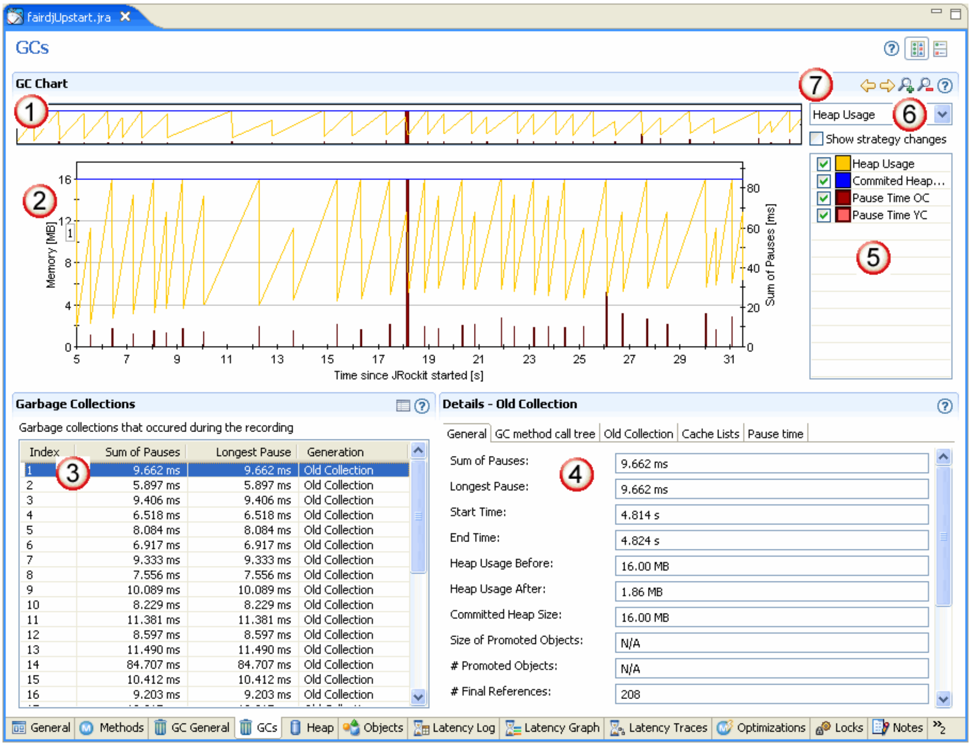
This section is divided into the following topics:

- Getting Familiar with the GCs Tab
- Changing Focus on GC Chart
- Viewing Specifics about Garbage Collections
- Viewing the Detailed Information About the Garbage Collection

Getting Familiar with the GCs Tab

The **GCs** tab visualizes how and when a garbage collection has occurred during the running of the application (Figure 9-1). It also shows specific information for each garbage collection.

Figure 9-1 The GCs tab



Garbage Collections

Garbage collections that occured during the recording

Index	Sum of Pauses	Longest Pause	Generation
1	9.662 ms	9.662 ms	Old Collection
2	5.897 ms	5.897 ms	Old Collection
3	9.406 ms	9.406 ms	Old Collection
4	6.518 ms	6.518 ms	Old Collection
5	8.084 ms	8.084 ms	Old Collection
6	6.917 ms	6.917 ms	Old Collection
7	9.333 ms	9.333 ms	Old Collection
8	7.556 ms	7.556 ms	Old Collection
9	10.089 ms	10.089 ms	Old Collection
10	8.229 ms	8.229 ms	Old Collection
11	11.381 ms	11.381 ms	Old Collection
12	8.597 ms	8.597 ms	Old Collection
13	11.490 ms	11.490 ms	Old Collection
14	84.707 ms	84.707 ms	Old Collection
15	10.412 ms	10.412 ms	Old Collection
16	9.203 ms	9.203 ms	Old Collection

Details - Old Collection

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

Sum of Pauses: 9.662 ms
Longest Pause: 9.662 ms
Start Time: 4.814 s
End Time: 4.824 s
Heap Usage Before: 16.00 MB
Heap Usage After: 1.86 MB
Committed Heap Size: 16.00 MB
Size of Promoted Objects: N/A
Promoted Objects: N/A
Final References: 208

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

1. **GCs Overview** timeline—this timeline 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 GC 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 Chart, 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 9-1) 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 panel 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** panel 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. **Drop-down list** and **Show strategy changes**—the drop-down list allows you to toggle between the *Heap Usage* and the *References and finalizers* view on the **Heap Usage** chart. If you select Show strategy changes, you will see when JRockit has changed garbage collection strategy.
7. **Move** and **Zoom** buttons—these buttons are used with the **GCs** Timeline.

Changing Focus on GC Chart

Depending on how long your JRA recording is, the **GC 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.

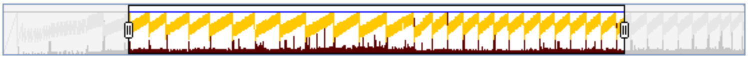
The two ways to refocus on the **GC Chart** are described here:

- To change focus on the Heap Usage chart
- To use the Move and Zoom buttons for the GC Chart

To change focus on the Heap Usage chart

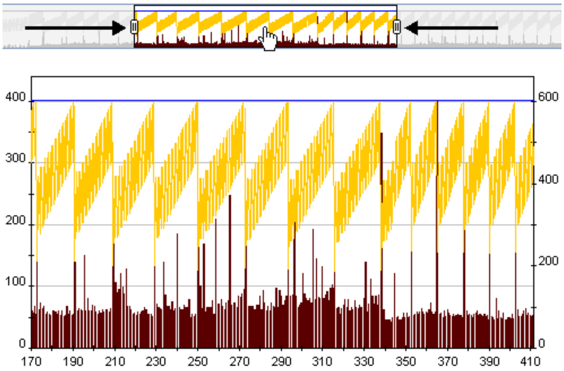
1. Click and drag the handles on both sides on the **GC Chart** (Figure 9-2).

Figure 9-2 The GC Chart zoom function



2. Drag the **GC Chart** into the desired position (Figure 9-3).

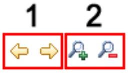
Figure 9-3 The GC Chart



To use the Move and Zoom buttons for the GC Chart

1. Click the **Move forward** or **Move backward** buttons (marked 1 in Figure 9-4) to first decrease the GC Chart view.

Figure 9-4 Move and Zoom buttons



2. Click either of the **Move** buttons to slide the focus on the GC Chart.
3. Click the **Zoom in** or **Zoom out** buttons (marked 2 in Figure 9-4) to decrease or increase the visible span of the GC Chart.

Viewing Specifics about Garbage Collections

The **Garbage Collections** table on the **GCs** 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. If you use the dynamic garbage 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). If you use a static garbage collector, there will not be any old or young collections. For more information on garbage collections, please see Garbage Collection in BEA JRockit in the BEA JRockit Diagnostics Guide.

This section is divided into the following topics:

- To view one garbage collection in the GC Chart
- To view many garbage collections in GC Chart

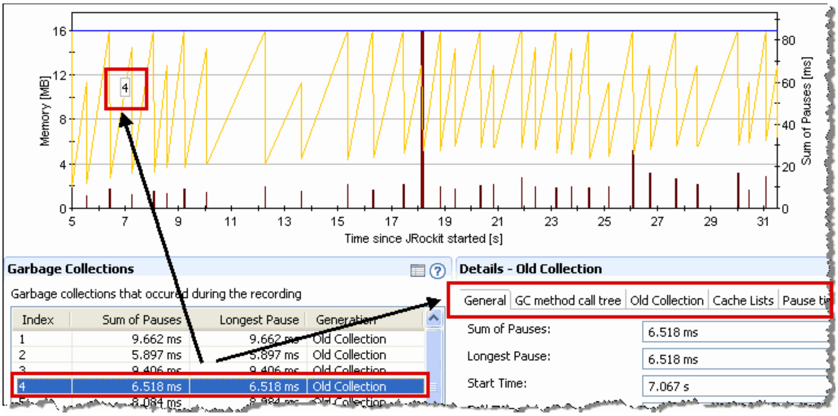
To view one garbage collection in the GC Chart

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** panel has also changed to show all the specifics about that garbage collection.

   The **Details** panel changes name depending on if the selected event is an old collection or a young collection (Figure 9-5).

Figure 9-5 Viewing one garbage collection



Garbage Collections

Garbage collections that occured during the recording

Index	Sum of Pauses	Longest Pause	Generation
1	9.662 ms	9.662 ms	Old Collection
2	5.897 ms	5.897 ms	Old Collection
3	9.406 ms	9.406 ms	Old Collection
4	6.518 ms	6.518 ms	Old Collection
5	8.084 ms	8.084 ms	Old Collection

Details - Old Collection

Sum of Pauses: 6.518 ms
Longest Pause: 6.518 ms
Start Time: 7.067 s

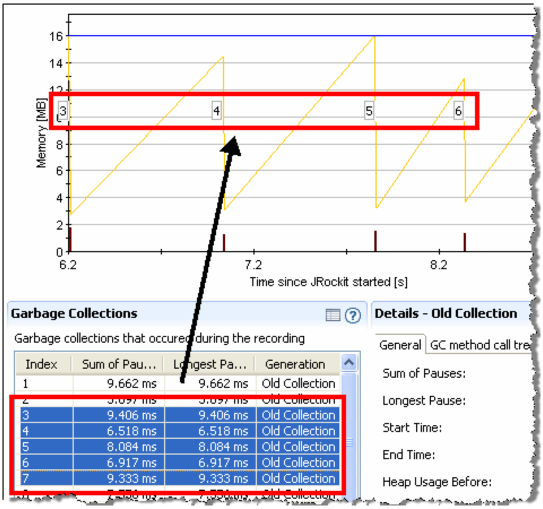
To view many garbage collections in GC Chart

1. Scroll 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** (Figure 9-6).

   Note: The garbage collection event that was last selected is the one that is displayed in the **Details** panel.

Figure 9-6 Viewing multiple garbage collections

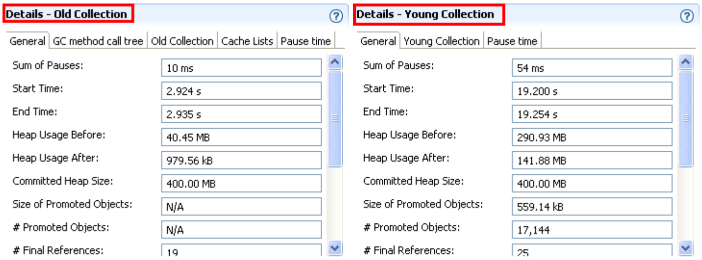


Index	Sum of Pau...	Longest Pa...	Generation
1	9.662 ms	9.662 ms	Old Collection
2	[illegible]	[illegible]	[illegible]
3	9.406 ms	9.406 ms	Old Collection
4	6.518 ms	6.518 ms	Old Collection
5	8.084 ms	8.084 ms	Old Collection
6	6.917 ms	6.917 ms	Old Collection
7	9.333 ms	9.333 ms	Old Collection

Viewing the Detailed Information About the Garbage Collection

When you select a garbage collection, the **Details** panel of the **GCs** tab changes name to either **Details - Old Collection** or **Details - Young Collection** depending on the type of garbage collection you have selected. You will also see different sets of tabs that contain specific information about the garbage collection that you have selected (Figure 9-7).

Figure 9-7 Tab differences when viewing old and young collections



	Details - Old Collection	Details - Young Collection
Sum of Pauses:	10 ms	54 ms
Start Time:	2.924 s	19.200 s
End Time:	2.935 s	19.254 s
Heap Usage Before:	40.45 MB	290.93 MB
Heap Usage After:	979.56 kB	141.88 MB
Committed Heap Size:	400.00 MB	400.00 MB
Size of Promoted Objects:	N/A	559.14 kB
# Promoted Objects:	N/A	17,144
# Final References:	19	25

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.

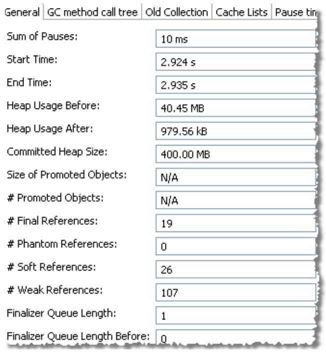
This section describes the following tabs:

- 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
- The Pause Time Tab

Viewing Information on the General Garbage Collection Tab

The **General** tab (Figure 9-8) displays information such as start time and end time of the garbage collection.

Figure 9-8 The General garbage collection tab



Sum of Pauses: 10 ms
Start Time: 2.924 s
End Time: 2.935 s
Heap Usage Before: 40.45 MB
Heap Usage After: 979.56 kB
Committed Heap Size: 400.00 MB
Size of Promoted Objects: N/A
Promoted Objects: N/A
Final References: 19
Phantom References: 0
Soft References: 26
Weak References: 107
Finalizer Queue Length: 1
Finalizer Queue Length Before: 0

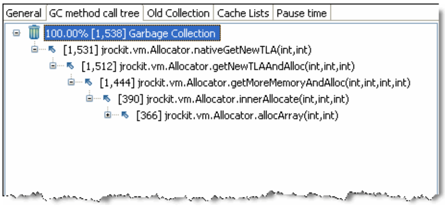
- **Sum of Pauses**—the sum of all pause times in milliseconds that the garbage collector stops all threads in JRockit. This is not the same as end time-start time in the case of a concurrent garbage collector.
- **Start/End Time**—the times when the garbage collection started and ended, counted in milliseconds from when JRockit started.
- **Heap Usage Before/After**—the used heap size before or after the garbage collection.
- **Committed Heap Size**—the total size of the heap (used plus unused memory) after the garbage collection.
- **Size of Promoted Objects** (and number of Promoted Objects)—the size (and the amount) of the objects that have been promoted to the old space.

- **References**—there are several types of references collected during a recording. For information on what a reference is, see Viewing Reference Objects in the Diagnostics Guide.
- **Finalizer Queue Length (and Before)**—the finalizer queue length.
- **Generation**—Indicates whether the garbage collector performed an old or young collection (see Generational Garbage Collection in the Diagnostics 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.

Viewing Information on the GC Method Call Tree Tab

The **GC Method Call Tree** tab (Figure 9-9) shows an aggregation of the call traces of the threads triggering a garbage collection.

Figure 9-9 The GC Method Call Tree tab

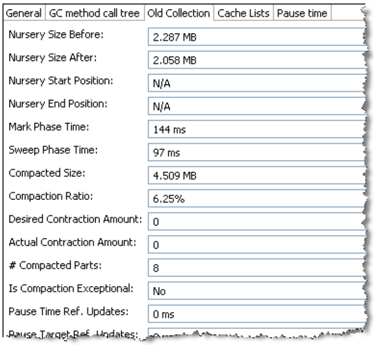


```
100.00% [1,538] Garbage Collection
  [1,531] jrockit.vm.Allocator.nativeGetNewTLA(int,int)
    [1,512] jrockit.vm.Allocator.getNewTLAAndAlloc(int,int,int)
      [1,444] jrockit.vm.Allocator.getMoreMemoryAndAlloc(int,int,int,int)
        [390] jrockit.vm.Allocator.innerAllocate(int,int,int)
          [366] jrockit.vm.Allocator.allocArray(int,int)
```

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** table. Here you find information about nursery, mark and sweep pause times, etc. (Figure 9-10).

Figure 9-10 The Old/Young Collection tab



Nursery Size Before: 2.287 MB
Nursery Size After: 2.058 MB
Nursery Start Position: N/A
Nursery End Position: N/A
Mark Phase Time: 144 ms
Sweep Phase Time: 97 ms
Compacted Size: 4.509 MB
Compaction Ratio: 6.25%
Desired Contraction Amount: 0
Actual Contraction Amount: 0
Compacted Parts: 8
Is Compaction Exceptional: No
Pause Time Ref. Updates: 0 ms
Pause Target Ref. Updates:

- **Nursery Size Before/After**—indicates the free space in the nursery before and the free space in the nursery after the garbage collection (in some cases the nursery size increases).

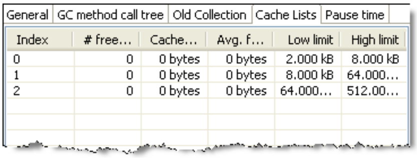
The information below is only valid for old collections:

- **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

The **Cache Lists** tab (Figure 9-11) displays the specification for the different cache lists. Each cache list contains settings for upper and lower cache size.

Figure 9-11 The Cache Lists tab

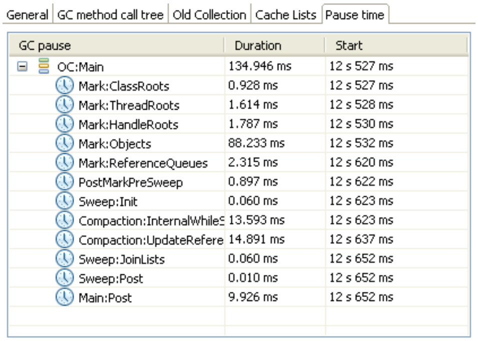


Index	# free...	Cache...	Avg. f...	Low limit	High limit
0	0	0 bytes	0 bytes	2.000 kB	8.000 kB
1	0	0 bytes	0 bytes	8.000 kB	64.000...
2	0	0 bytes	0 bytes	64.000...	512.00...

- **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.



GC pause	Duration	Start
⊟ OC:Main	134.946 ms	12 s 527 ms
Mark:ClassRoots	0.928 ms	12 s 527 ms
Mark:ThreadRoots	1.614 ms	12 s 528 ms
Mark:HandleRoots	1.787 ms	12 s 530 ms
Mark:Objects	88.233 ms	12 s 532 ms
Mark:ReferenceQueues	2.315 ms	12 s 620 ms
PostMarkPreSweep	0.897 ms	12 s 622 ms
Sweep:Init	0.060 ms	12 s 623 ms
Compaction:InternalWhileS	13.593 ms	12 s 623 ms
Compaction:UpdateRefere	14.891 ms	12 s 637 ms
Sweep:JoinLists	0.060 ms	12 s 652 ms
Sweep:Post	0.010 ms	12 s 652 ms
Main:Post	9.926 ms	12 s 652 ms

- **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.

- **Duration**—this is the length, measured in milliseconds, of the pause.
- **Start/End**—this is the start and end time, measured in milliseconds. You can change how the time is displayed by right-clicking in the table and select **Start** and then the value for the time.

Java Heap Content Information

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

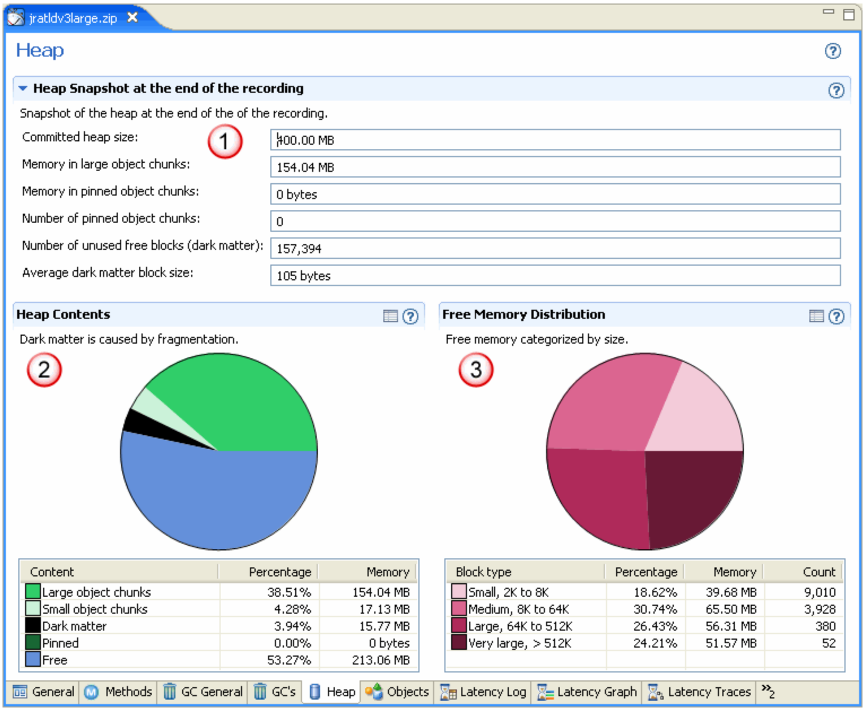
This section contains the following topics:

- Getting Familiar with the Heap 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 Tab

The **Heap** tab depicts Java heap contents and free memory distribution (Figure 10-1).

Figure 10-1 The Heap tab



Snapshot of the heap at the end of the of the recording.

Committed heap size:	400.00 MB
Memory in large object chunks:	154.04 MB
Memory in pinned object chunks:	0 bytes
Number of pinned object chunks:	0
Number of unused free blocks (dark matter):	157,394
Average dark matter block size:	105 bytes

Dark matter is caused by fragmentation.

Content	Percentage	Memory
Large object chunks	38.51%	154.04 MB
Small object chunks	4.28%	17.13 MB
Dark matter	3.94%	15.77 MB
Pinned	0.00%	0 bytes
Free	53.27%	213.06 MB

Free memory categorized by size.

Block type	Percentage	Memory	Count
Small, 2K to 8K	18.62%	39.68 MB	9,010
Medium, 8K to 64K	30.74%	65.50 MB	3,928
Large, 64K to 512K	26.43%	56.31 MB	380
Very large, > 512K	24.21%	51.57 MB	52

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

1. **Heap Snapshot at the End of the Recording**—this panel 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.

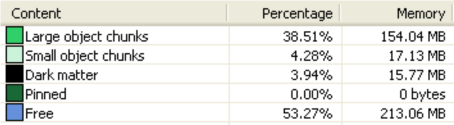
Viewing the Heap Contents Information

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 on the Java heap that is wasted due to fragmentation. It is normal to have a certain amount of dark matter on the heap.

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

The table below the pie chart (Figure 10-2) lists all objects with the exact data: memory in MB and percentage that they occupy of the heap.

Figure 10-2 Heap content table

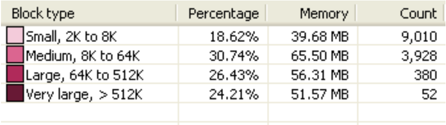


Content	Percentage	Memory
🟩 Large object chunks	38.51%	154.04 MB
🟩 Small object chunks	4.28%	17.13 MB
⬛ Dark matter	3.94%	15.77 MB
🟩 Pinned	0.00%	0 bytes
🟦 Free	53.27%	213.06 MB

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 table below the pie chart (Figure 10-3) lists all block sizes by category.

Figure 10-3 Free memory content table



Block type	Percentage	Memory	Count
■ Small, 2K to 8K	18.62%	39.68 MB	9,010
■ Medium, 8K to 64K	30.74%	65.50 MB	3,928
■ Large, 64K to 512K	26.43%	56.31 MB	380
■ Very large, > 512K	24.21%	51.57 MB	52

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

Objects Information

The **Objects** tab displays the most common types and classes occupying the Java heap at the beginning and at the end of the JRA recording.

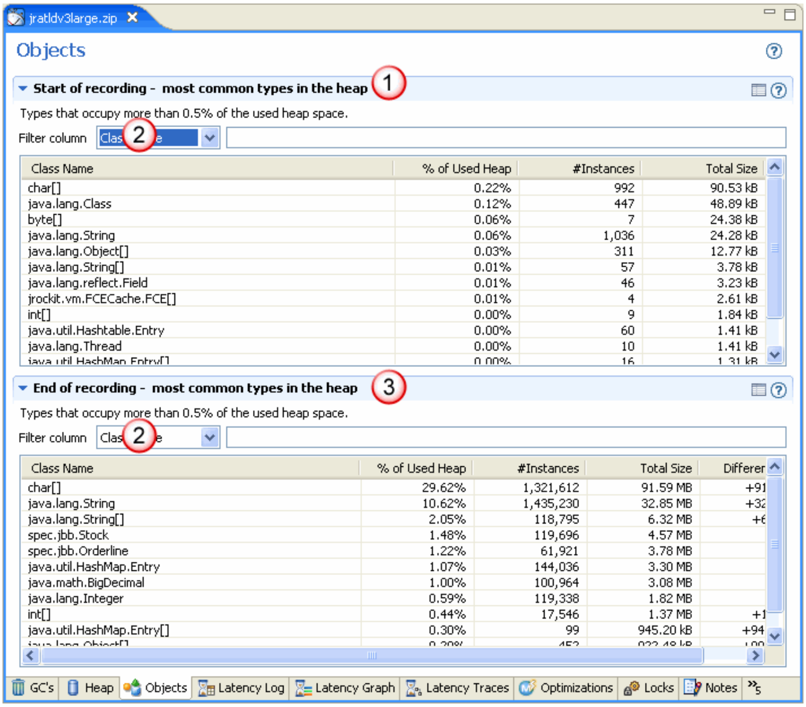
This section is divided into the following topics:

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

Getting Familiar with the Objects 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** tab (Figure 11-1). Abnormal results in the object statistics might help you detect the existence of a memory leak in your application.

Figure 11-1 The Objects tab



Objects

Start of recording - most common types in the heap

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

Filter column Clas e

Class Name	% of Used Heap	#Instances	Total Size
char[]	0.22%	992	90.53 kB
java.lang.Class	0.12%	447	48.89 kB
byte[]	0.06%	7	24.38 kB
java.lang.String	0.06%	1,036	24.28 kB
java.lang.Object[]	0.03%	311	12.77 kB
java.lang.String[]	0.01%	57	3.78 kB
java.lang.reflect.Field	0.01%	46	3.23 kB
jrockit.vm.FCECache.FCE[]	0.01%	4	2.61 kB
int[]	0.00%	9	1.84 kB
java.util.Hashtable.Entry	0.00%	60	1.41 kB
java.lang.Thread	0.00%	10	1.41 kB

End of recording - most common types in the heap

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

Filter column Clas e

Class Name	% of Used Heap	#Instances	Total Size	Differer
char[]	29.62%	1,321,612	91.59 MB	+91
java.lang.String	10.62%	1,435,230	32.85 MB	+32
java.lang.String[]	2.05%	118,795	6.32 MB	+6
spec.jbb.Stock	1.48%	119,696	4.57 MB	
spec.jbb.Orderline	1.22%	61,921	3.78 MB	
java.util.HashMap.Entry	1.07%	144,036	3.30 MB	
java.math.BigDecimal	1.00%	100,964	3.08 MB	
java.lang.Integer	0.59%	119,338	1.82 MB	
int[]	0.44%	17,546	1.37 MB	+1
java.util.HashMap.Entry[]	0.30%	99	945.20 kB	+94

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

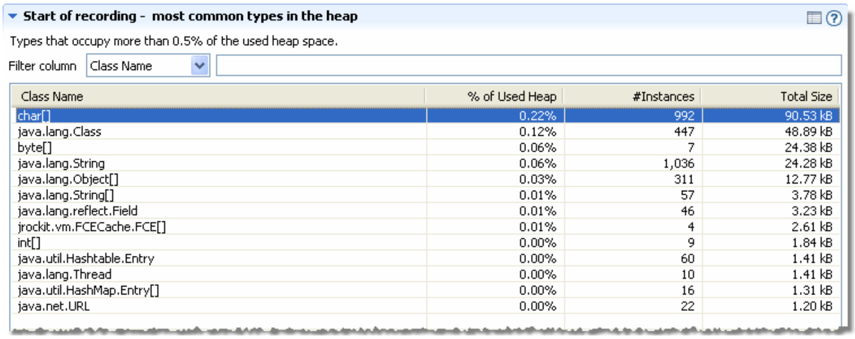
1. **Start of Recording**—this table lists the most common types on the heap at the beginning of the recording.
2. **Filter column**—see Filtering Information on how to use this function.
3. **End of Recording**—this table lists the most common types on the heap at the end of the recording.

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** table (Figure 11-2).

Use the filtering function to find the object you are looking for, see Filtering Information.

Figure 11-2 Start of Recording table



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

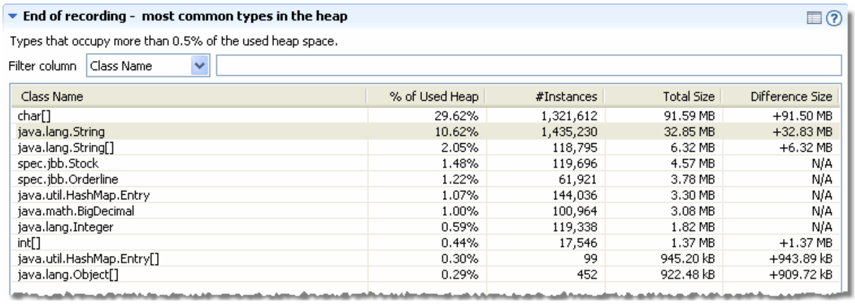
Class Name	% of Used Heap	#Instances	Total Size
char[]	0.22%	992	90.53 kB
java.lang.Class	0.12%	447	48.89 kB
byte[]	0.06%	7	24.38 kB
java.lang.String	0.06%	1,036	24.28 kB
java.lang.Object[]	0.03%	311	12.77 kB
java.lang.String[]	0.01%	57	3.78 kB
java.lang.reflect.Field	0.01%	46	3.23 kB
jrockit.vm.FCECache.FCE[]	0.01%	4	2.61 kB
int[]	0.00%	9	1.84 kB
java.util.Hashtable.Entry	0.00%	60	1.41 kB
java.lang.Thread	0.00%	10	1.41 kB
java.util.HashMap.Entry[]	0.00%	16	1.31 kB
java.net.URL	0.00%	22	1.20 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** table (Figure 11-3).

Use the filtering function to find the object you are looking for, see Filtering Information.

Figure 11-3 End of Recording table



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[]	29.62%	1,321,612	91.59 MB	+91.50 MB
java.lang.String	10.62%	1,435,230	32.85 MB	+32.83 MB
java.lang.String[]	2.05%	118,795	6.32 MB	+6.32 MB
spec.jbb.Stock	1.48%	119,696	4.57 MB	N/A
spec.jbb.Orderline	1.22%	61,921	3.78 MB	N/A
java.util.HashMap.Entry	1.07%	144,036	3.30 MB	N/A
java.math.BigDecimal	1.00%	100,964	3.08 MB	N/A
java.lang.Integer	0.59%	119,338	1.82 MB	N/A
int[]	0.44%	17,546	1.37 MB	+1.37 MB
java.util.HashMap.Entry[]	0.30%	99	945.20 kB	+943.89 kB
java.lang.Object[]	0.29%	452	922.48 kB	+909.72 kB

Code Optimization Information

JRockit continuously look for ways to optimize code. The **Optimizations** tab displays the methods that were optimized by the adaptive optimization system in JRockit during the recording.

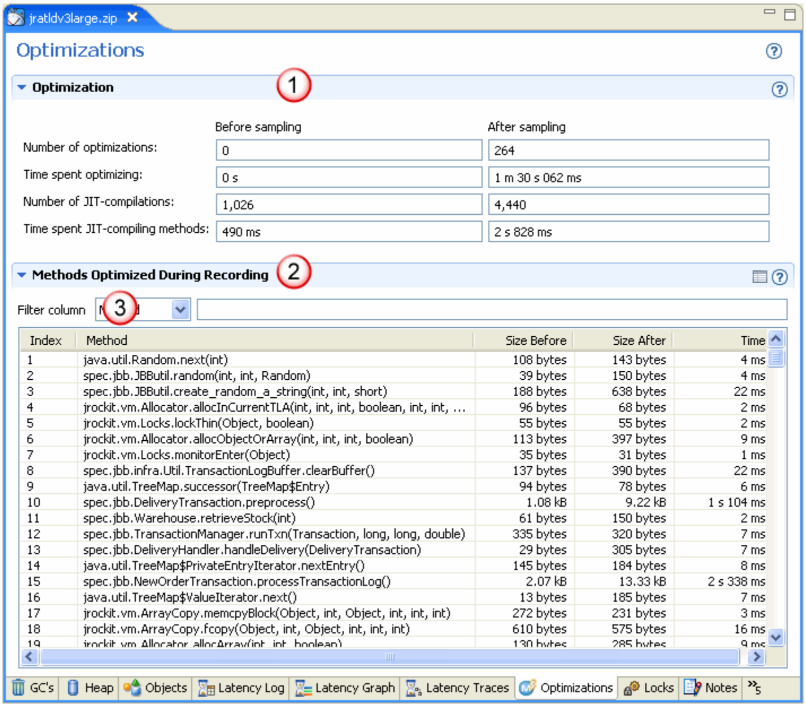
This section is divided into the following topics:

- Getting Familiar with the Optimizations Tab
- Viewing Optimization Information
- Viewing Methods Optimized During Recording Information

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 (Figure 12-1).

Figure 12-1 Optimizations tab



	Before sampling	After sampling
Number of optimizations:	0	264
Time spent optimizing:	0 s	1 m 30 s 062 ms
Number of JIT-compilations:	1,026	4,440
Time spent JIT-compiling methods:	490 ms	2 s 828 ms

Filter column

Index	Method	Size Before	Size After	Time
1	java.util.Random.next(int)	108 bytes	143 bytes	4 ms
2	spec.jbb.JBButil.random(int, int, Random)	39 bytes	150 bytes	4 ms
3	spec.jbb.JBButil.create_random_a_string(int, int, short)	188 bytes	638 bytes	22 ms
4	jrockit.vm.Allocator.allocInCurrentTLA(int, int, int, boolean, int, int, ...	96 bytes	68 bytes	2 ms
5	jrockit.vm.Locks.lockThin(Object, boolean)	55 bytes	55 bytes	2 ms
6	jrockit.vm.Allocator.allocObjectOrArray(int, int, int, boolean)	113 bytes	397 bytes	9 ms
7	jrockit.vm.Locks.monitorEnter(Object)	35 bytes	31 bytes	1 ms
8	spec.jbb.infra.Util.TransactionLogBuffer.clearBuffer()	137 bytes	390 bytes	22 ms
9	java.util.TreeMap.successor(TreeMap$Entry)	94 bytes	78 bytes	6 ms
10	spec.jbb.DeliveryTransaction.preprocess()	1.08 kB	9.22 kB	1 s 104 ms
11	spec.jbb.Warehouse.retrieveStock(int)	61 bytes	150 bytes	2 ms
12	spec.jbb.TransactionManager.runTxn(Transaction, long, long, double)	335 bytes	320 bytes	7 ms
13	spec.jbb.DeliveryHandler.handleDelivery(DeliveryTransaction)	29 bytes	305 bytes	7 ms
14	java.util.TreeMap$PrivateEntryIterator.nextEntry()	145 bytes	184 bytes	8 ms
15	spec.jbb.NewOrderTransaction.processTransactionLog()	2.07 kB	13.33 kB	2 s 338 ms
16	java.util.TreeMap$ValueIterator.next()	13 bytes	185 bytes	7 ms
17	jrockit.vm.ArrayCopy.memcpyBlock(Object, int, Object, int, int, int)	272 bytes	231 bytes	3 ms
18	jrockit.vm.ArrayCopy.fcopy(Object, int, Object, int, int, int)	610 bytes	575 bytes	16 ms
19	jrockit.vm.Allocator.allocArray(int, int, boolean)	130 bytes	285 bytes	9 ms

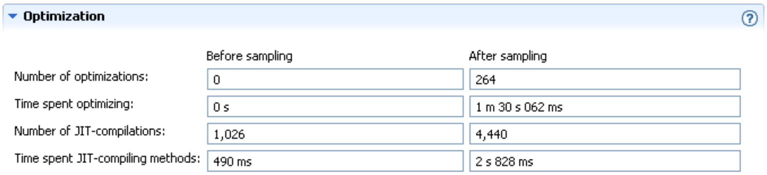
The **Optimizations** tab is divided into the following sections:

1. **Optimization**—this panel displays the before and after scenario of the optimizations that have taken place.
2. **Methods Optimized During Recording**—this table lists 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.
3. **Filter column**—see Filtering Information on how to use this function.

Viewing Optimization Information

The **Optimization** panel (Figure 12-2) 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 12-2 Optimization panel



	Before sampling	After sampling
Number of optimizations:	0	264
Time spent optimizing:	0 s	1 m 30 s 062 ms
Number of JIT-compilations:	1,026	4,440
Time spent JIT-compiling methods:	490 ms	2 s 828 ms

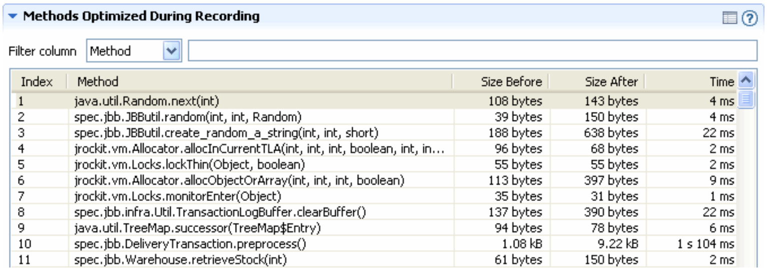
Viewing Methods Optimized During Recording Information

The **Methods Optimized During Recording** table (Figure 12-3) 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.

Use the filtering function to find the method you are looking for, see Filtering Information.

Figure 12-3 Methods Optimized During Recording table



Filter column: Method

Index	Method	Size Before	Size After	Time
1	java.util.Random.next(int)	108 bytes	143 bytes	4 ms
2	spec.jbb.JBButil.random(int, int, Random)	39 bytes	150 bytes	4 ms
3	spec.jbb.JBButil.create_random_a_string(int, int, short)	188 bytes	638 bytes	22 ms
4	jrockit.vm.Allocator.allocInCurrentTLA(int, int, int, boolean, int, in...	96 bytes	68 bytes	2 ms
5	jrockit.vm.Locks.lockThin(Object, boolean)	55 bytes	55 bytes	2 ms
6	jrockit.vm.Allocator.allocObjectOrArray(int, int, int, boolean)	113 bytes	397 bytes	9 ms
7	jrockit.vm.Locks.monitorEnter(Object)	35 bytes	31 bytes	1 ms
8	spec.jbb.infra.Util.TransactionLogBuffer.clearBuffer()	137 bytes	390 bytes	22 ms
9	java.util.TreeMap.successor(TreeMap$Entry)	94 bytes	78 bytes	6 ms
10	spec.jbb.DeliveryTransaction.preprocess()	1.08 kB	9.22 kB	1 s 104 ms
11	spec.jbb.Warehouse.retrieveStock(int)	61 bytes	150 bytes	2 ms

Lock Profiling Information

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

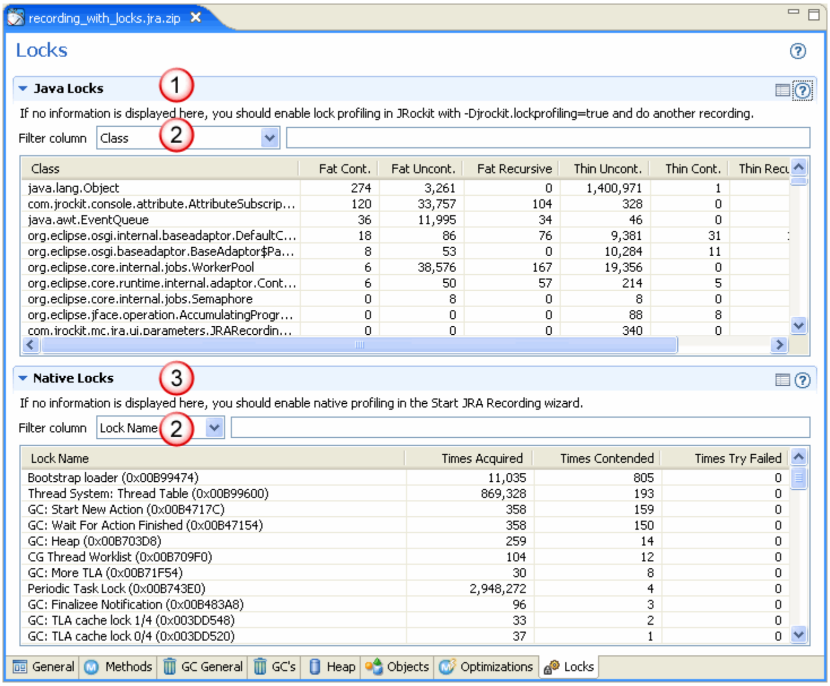
This section is divided into the following topics:

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

Getting Familiar with the Locks Tab

The **Locks** tab displays lock information for both your application and JRockit (Figure 13-1).

Figure 13-1 Locks tab



If no information is displayed here, you should enable lock profiling in JRockit with -Djrockit.lockprofiling=true and do another recording.

Filter column: Class

Class	Fat Cont.	Fat Uncont.	Fat Recursive	Thin Uncont.	Thin Cont.	Thin Recu
java.lang.Object	274	3,261	0	1,400,971	1	
com.jrockit.console.attribute.AttributeSubscrip…	120	33,757	104	328	0	
java.awt.EventQueue	36	11,995	34	46	0	
org.eclipse.osgi.internal.baseadaptor.DefaultC…	18	86	76	9,381	31	
org.eclipse.osgi.baseadaptor.BaseAdaptor$Pa…	8	53	0	10,284	11	
org.eclipse.core.internal.jobs.WorkerPool	6	38,576	167	19,356	0	
org.eclipse.core.runtime.internal.adaptor.Cont…	6	50	57	214	5	
org.eclipse.core.internal.jobs.Semaphore	0	8	0	8	0	
org.eclipse.jface.operation.AccumulatingProgr…	0	0	0	88	8	
com.jrockit.mc.jra.ui.parameters.JRARecordin…	0	0	0	340	0	

If no information is displayed here, you should enable native profiling in the Start JRA Recording wizard.

Filter column: Lock Name

Lock Name	Times Acquired	Times Contended	Times Try Failed
Bootstrap loader (0x00B99474)	11,035	805	0
Thread System: Thread Table (0x00B99600)	869,328	193	0
GC: Start New Action (0x00B4717C)	358	159	0
GC: Wait For Action Finished (0x00B47154)	358	150	0
GC: Heap (0x00B703D8)	259	14	0
CG Thread Worklist (0x00B709F0)	104	12	0
GC: More TLA (0x00B71F54)	30	8	0
Periodic Task Lock (0x00B743E0)	2,948,272	4	0
GC: Finalizee Notification (0x00B483A8)	96	3	0
GC: TLA cache lock 1/4 (0x003DD548)	33	2	0
GC: TLA cache lock 0/4 (0x003DD520)	37	1	0

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

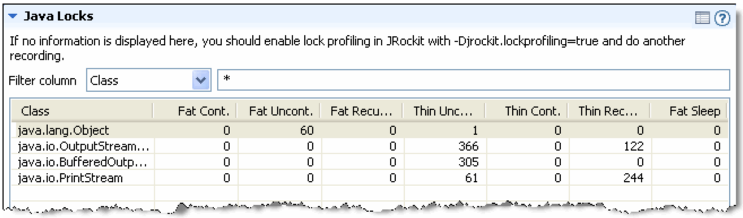
1. **Java Locks**—this table lists all locks in your application.
2. **Filter column**—see Filtering Information on how to use this function.
3. **Native Locks**—this table lists all locks in JRockit.

Java Locks Profiling

The information that is displayed under the **Java Locks** chart (Figure 13-2) 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. For more information on locks, please refer to About Thin, Fat, Recursive, and Contended Locks in BEA JRockit.

Use the filtering function to find the Java locks you are looking for, see Filtering Information.

Figure 13-2 Java Locks



If no information is displayed here, you should enable lock profiling in JRockit with -Djrockit.lockprofiling=true and do another recording.

Filter column: Class | *

Class	Fat Cont.	Fat Uncont.	Fat Recu...	Thin Unc...	Thin Cont.	Thin Rec...	Fat Sleep
java.lang.Object	0	60	0	1	0	0	0
java.io.OutputStream...	0	0	0	366	0	122	0
java.io.BufferedOutp...	0	0	0	305	0	0	0
java.io.PrintStream	0	0	0	61	0	244	0

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 table is blank, it is not enabled.

To enable Java lock profiling data

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

  For example:

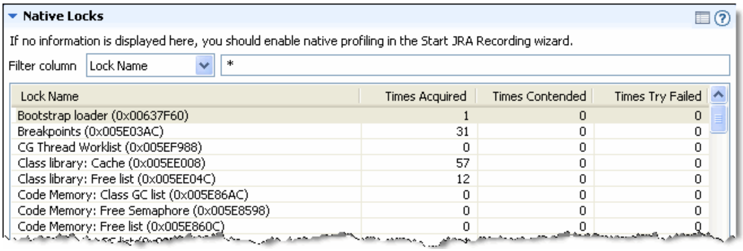
```
java -Djrockit.lockprofiling=true -XXjra:<AnyJRAParam> -jar MyApplication.jar
```

Native Lock Profiling

If you are looking at a recording of JRockit J2SE 5.0 or later, the recording includes information about native locks (Figure 13-3). Native locks are locks in the JRockit internal code and is nothing your application can control.

Use the filtering function to find the Java locks you are looking for, see Filtering Information.

Figure 13-3 Native Locks



If no information is displayed here, you should enable native profiling in the Start JRA Recording wizard.

Filter column: Lock Name | *

Lock Name	Times Acquired	Times Contended	Times Try Failed
Bootstrap loader (0x00637F60)	1	0	0
Breakpoints (0x005E03AC)	31	0	0
CG Thread Worklist (0x005EF988)	0	0	0
Class library: Cache (0x005EE008)	57	0	0
Class library: Free list (0x005EE04C)	12	0	0
Code Memory: Class GC list (0x005E86AC)	0	0	0
Code Memory: Free Semaphore (0x005E8598)	0	0	0
Code Memory: Free list (0x005E860C)	0	0	0

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 **Locks** tab, the native sampling was not enabled during the recording. See 3. Create JRA Recording for information on how to enable native sampling.

Start and End Processes Information

The **Processes** tab lists which processes were running during the start and the end of the JRA recording. The information found on this tab is mostly geared towards engineers within the BEA customer support (Customer Centric Engineering, CCE) team. CCE uses the information to get a picture of which applications that were running on the machine when, for example, a crash has occurred. This tab is not visible by default, so you need to turn it on before you can view that information.

This section is divided into the following topics:

- Turning on the Processes Tab
- Getting Familiar with the Processes Tab
- Snapshot of Processes at Beginning and End of Recording
- Detailed Processes Information

Turning on the Processes Tab

The **Process** tab is not visible unless you have selected the option **Enable extra information for JRPG CCE** in the JRA Preferences

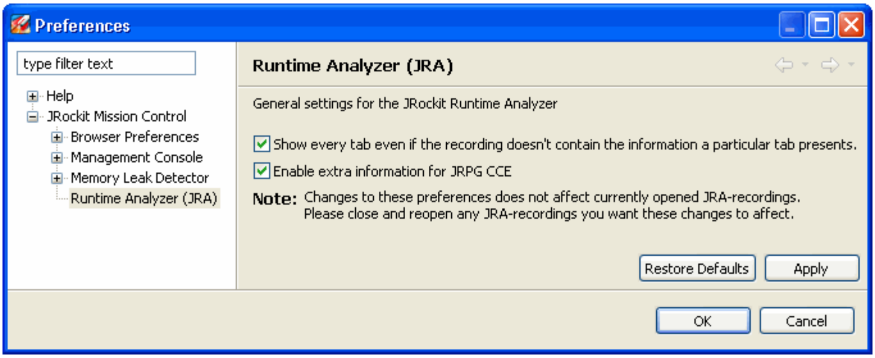
To turn on the Processes tab

1. Click **Window** > **Preferences**.

   The **Preferences** window appears.

2. Click **JRockit Mission Control > Runtime Analyzer (JRA)**.
3. Select the **Enable extra information for JRPG CCE** option (Figure 14-1).

Figure 14-1 Preferences window



Runtime Analyzer (JRA)

General settings for the JRockit Runtime Analyzer

☑ Show every tab even if the recording doesn't contain the information a particular tab presents.

☑ Enable extra information for JRPG CCE

Note: Changes to these preferences does not affect currently opened JRA-recordings. Please close and reopen any JRA-recordings you want these changes to affect.

4. Click **Apply**.
5. Click **OK**.

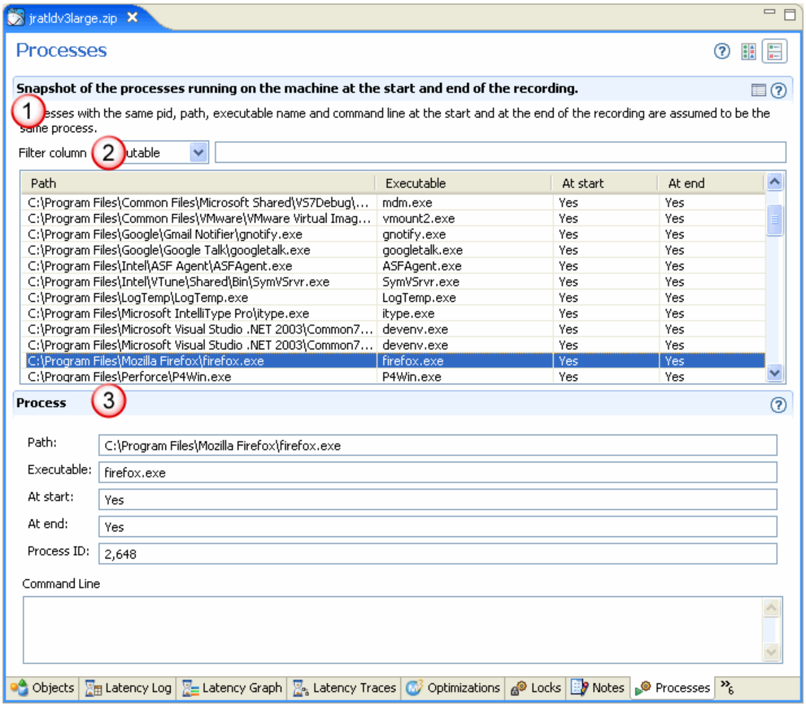
   If you have a JRA recording open when you change this preference, you need to close it and then open it again for the **Processes** tab to become visible.

Getting Familiar with the Processes Tab

The **Processes** tab displays start and end information of running processes (Figure 14-2).

Note: You need to enable the **Processes** tab for it to be visible in the JRA Tool (see To turn on the Processes tab).

Figure 14-2 Processes tab



Snapshot of the processes running on the machine at the start and end of the recording.

esses with the same pid, path, executable name and command line at the start and at the end of the recording are assumed to be the same process.

Path	Executable	At start	At end
C:\Program Files\Common Files\Microsoft Shared\VS7Debug\...	mdm.exe	Yes	Yes
C:\Program Files\Common Files\VMware\VMware Virtual Imag...	vmount2.exe	Yes	Yes
C:\Program Files\Google\Gmail Notifier\gnotify.exe	gnotify.exe	Yes	Yes
C:\Program Files\Google\Google Talk\googletalk.exe	googletalk.exe	Yes	Yes
C:\Program Files\Intel\ASF Agent\ASFAgent.exe	ASFAgent.exe	Yes	Yes
C:\Program Files\Intel\VTune\Shared\Bin\SymVSrvr.exe	SymVSrvr.exe	Yes	Yes
C:\Program Files\LogTemp\LogTemp.exe	LogTemp.exe	Yes	Yes
C:\Program Files\Microsoft IntelliType Pro\itype.exe	itype.exe	Yes	Yes
C:\Program Files\Microsoft Visual Studio .NET 2003\Common7...	devenv.exe	Yes	Yes
C:\Program Files\Microsoft Visual Studio .NET 2003\Common7...	devenv.exe	Yes	Yes
C:\Program Files\Mozilla Firefox\firefox.exe	firefox.exe	Yes	Yes
C:\Program Files\Perforce\P4Win.exe	P4Win.exe	Yes	Yes

Process

Path: C:\Program Files\Mozilla Firefox\firefox.exe
Executable: firefox.exe
At start: Yes
At end: Yes
Process ID: 2,648

Command Line

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

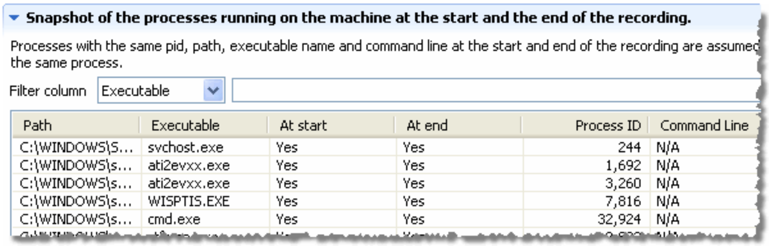
1. **Snapshot of the processes running on the machine at the start and at the end of the recording**—this table lists all processes that were active either during the start or the end of the recording or both.
2. **Filter column**—see Filtering Information on how to use this function.
3. **Process**—this panel details the processes information.

Snapshot of Processes at Beginning and End of Recording

The information that is displayed under the **Snapshot** view (Figure 14-3) lists all processes that were running at the start of the recording and at the end of the recording.

Use the filtering function to find the process you are looking for, see Filtering Information.

Figure 14-3 Snapshot view



Snapshot of the processes running on the machine at the start and the end of the recording.

Processes with the same pid, path, executable name and command line at the start and end of the recording are assumed the same process.

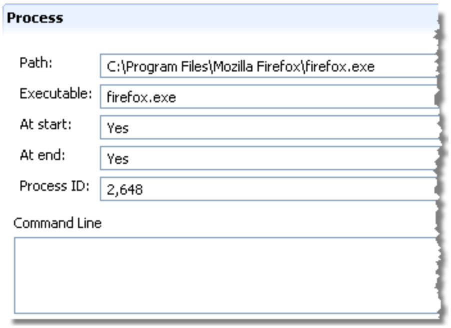
Filter column: Executable

Path	Executable	At start	At end	Process ID	Command Line
C:\WINDOWS\S...	svchost.exe	Yes	Yes	244	N/A
C:\WINDOWS\s...	ati2evxx.exe	Yes	Yes	1,692	N/A
C:\WINDOWS\s...	ati2evxx.exe	Yes	Yes	3,260	N/A
C:\WINDOWS\s...	WISPTIS.EXE	Yes	Yes	7,816	N/A
C:\WINDOWS\s...	cmd.exe	Yes	Yes	32,924	N/A

Detailed Processes Information

When selecting a process in the **Snapshot** view, you see a list of all details for that process at the bottom of the tab (Figure 14-4). 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 14-4 Detail process view



Process

Path: C:\Program Files\Mozilla Firefox\firefox.exe

Executable: firefox.exe

At start: Yes

At end: Yes

Process ID: 2,648

Command Line

CHAPTER 15

Threads Information

The **Threads** tab lists all thread dumps that have been taken during the recording. If no Thread Dump interval (in the recording options) is specified, the recording will contain a thread dump from the start and the end of the recording. A thread dump reveals information about an application's thread activity that can help you diagnose problems and better optimize application and JVM performance; for example, thread dumps automatically show the occurrence of a deadlock. Deadlocks bring some or all of an application to a complete halt.

The information found on this tab is mostly geared towards engineers within the BEA customer support (Customer Centric Engineering, CCE) team. This tab is not visible by default, so you need to turn it on before you can view that information.

Note: For comprehensive information on how create and use a thread dump, please see the Using Thread Dumps section in the BEA JRockit Diagnostics Guide.

This section is divided into the following topics:

- Turning on the Threads Tab
- Getting Familiar with the Threads Tab
- List of Times when Thread Dump is Taken
- Thread Dump Information

Turning on the Threads Tab

The **Threads** tab is not visible unless you have selected the option **Enable extra information for JRPG CCE** in the JRA Preferences.

To turn on the Threads tab

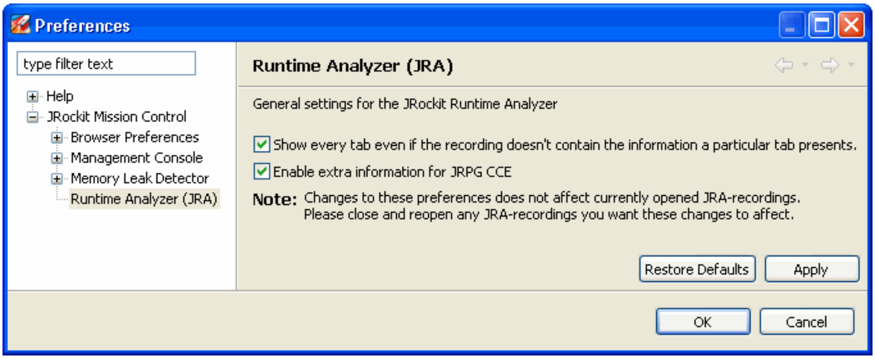
1. Click **Window > Preferences**.

   The **Preferences** window appears.

2. Click **JRockit Mission Control > Runtime Analyzer (JRA)**.

3. Select the **Enable extra information for JRPG CCE** option (Figure 15-1).

Figure 15-1 Preferences window



Runtime Analyzer (JRA)

General settings for the JRockit Runtime Analyzer

☑ Show every tab even if the recording doesn't contain the information a particular tab presents.

☑ Enable extra information for JRPG CCE

Note: Changes to these preferences does not affect currently opened JRA-recordings. Please close and reopen any JRA-recordings you want these changes to affect.

4. Click **Apply**.

5. Click **OK**.

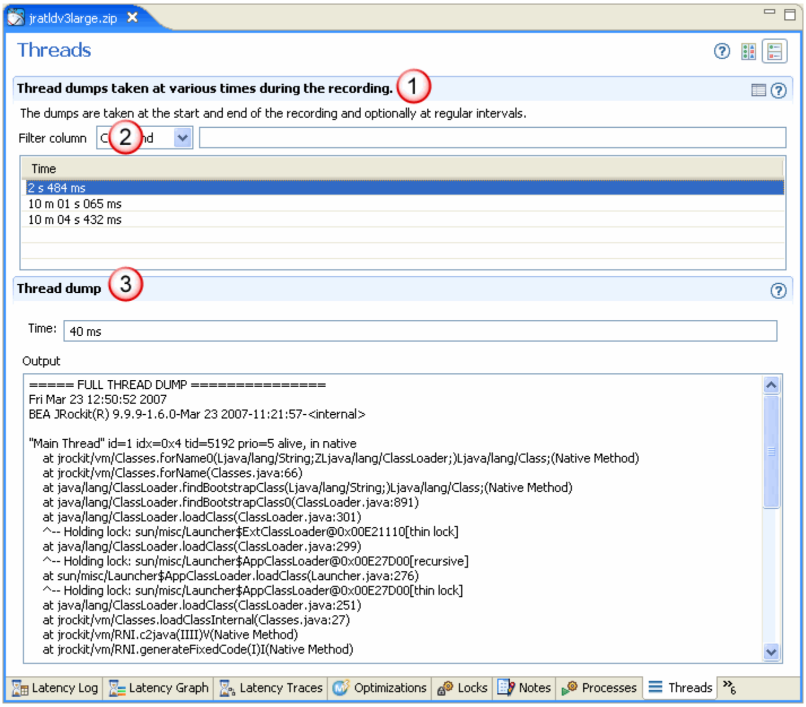
   If you have a JRA recording open when you change this preference, you need to close it and then open it again for the **Threads** tab to become visible.

Getting Familiar with the Threads Tab

The **Threads** tab lists the available thread dumps and by clicking on a specific time when a thread dump was created, you see the entire thread dump in the Thread dump (Figure 15-2).

Note: You need to enable the **Threads** tab for it to be visible in the JRA Tool (see To turn on the Threads tab).

Figure 15-2 Threads tab



Thread dumps taken at various times during the recording.

The dumps are taken at the start and end of the recording and optionally at regular intervals.

Filter column: Command

Time
2 s 484 ms
10 m 01 s 065 ms
10 m 04 s 432 ms

Thread dump

Time: 40 ms

Output

```
===== FULL THREAD DUMP ===============
Fri Mar 23 12:50:52 2007
BEA JRockit(R) 9.9.9-1.6.0-Mar 23 2007-11:21:57-<internal>

"Main Thread" id=1 idx=0x4 tid=5192 prio=5 alive, in native
    at jrockit/vm/Classes.forName0(Ljava/lang/String;ZLjava/lang/ClassLoader;)Ljava/lang/Class;(Native Method)
    at jrockit/vm/Classes.forName(Classes.java:66)
    at java/lang/ClassLoader.findBootstrapClass(Ljava/lang/String;)Ljava/lang/Class;(Native Method)
    at java/lang/ClassLoader.findBootstrapClass0(ClassLoader.java:891)
    at java/lang/ClassLoader.loadClass(ClassLoader.java:301)
    ^-- Holding lock: sun/misc/Launcher$ExtClassLoader@0x00E21110[thin lock]
    at java/lang/ClassLoader.loadClass(ClassLoader.java:299)
    ^-- Holding lock: sun/misc/Launcher$AppClassLoader@0x00E27D00[recursive]
    at sun/misc/Launcher$AppClassLoader.loadClass(Launcher.java:276)
    ^-- Holding lock: sun/misc/Launcher$AppClassLoader@0x00E27D00[thin lock]
    at java/lang/ClassLoader.loadClass(ClassLoader.java:251)
    at jrockit/vm/Classes.loadClassInternal(Classes.java:27)
    at jrockit/vm/RNI.c2java(IIII)V(Native Method)
    at jrockit/vm/RNI.generateFixedCode(I)I(Native Method)
```

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

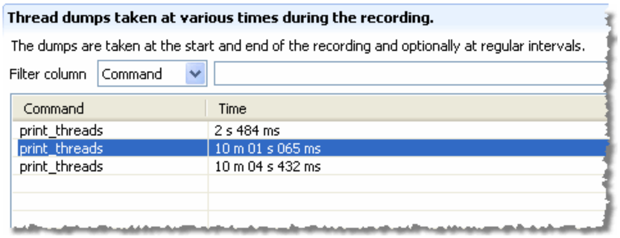
1. **Thread dumps taken at various times during the recording**—this table lists the times when a thread dump has been taken.
2. **Filter column**—see Filtering Information on how to use this function.
3. **Thread dump**—this panel displays the actual content of the selected thread dump.

List of Times when Thread Dump is Taken

The information that is displayed under the **Thread dumps taken at various times during the recording** table (Figure 15-3) shows when a thread dump was taken. You set the interval for taking thread dumps under the advanced option when you create a JRA recording (see To use the normal recording profile with advanced options).

Use the filtering function to find the specific thread dump, see Filtering Information.

Figure 15-3 Time view



Thread dumps taken at various times during the recording.

The dumps are taken at the start and end of the recording and optionally at regular intervals.

Filter column: Command

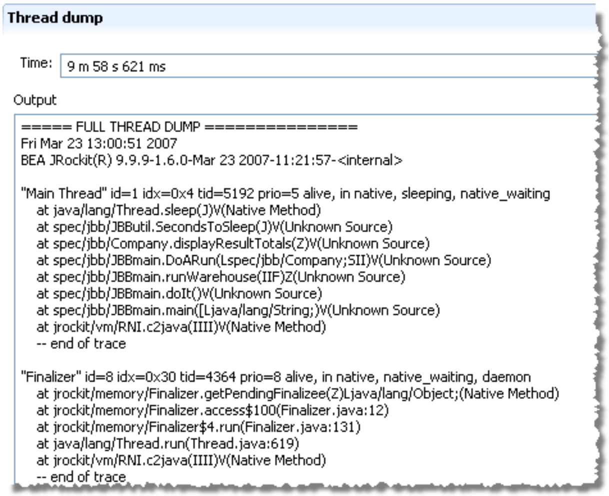
Command	Time
print_threads	2 s 484 ms
print_threads	10 m 01 s 065 ms
print_threads	10 m 04 s 432 ms

Thread Dump Information

When selecting a thread dump in the **Thread dump** list view, the entire thread dump is displayed in the **Thread dump** panel (Figure 15-4).

Note: To understand the information in the thread dump, please see the Using Thread Dumps section in the BEA JRockit Diagnostics Guide.

Figure 15-4 Thread dump output



Thread dump

Time: 9 m 58 s 621 ms

Output

```
===== FULL THREAD DUMP ================
Fri Mar 23 13:00:51 2007
BEA JRockit(R) 9.9.9-1.6.0-Mar 23 2007-11:21:57-<internal>

"Main Thread" id=1 idx=0x4 tid=5192 prio=5 alive, in native, sleeping, native_waiting
    at java/lang/Thread.sleep(J)V(Native Method)
    at spec/jbb/JBButil.SecondsToSleep(J)V(Unknown Source)
    at spec/jbb/Company.displayResultTotals(Z)V(Unknown Source)
    at spec/jbb/JBBmain.DoARun(Lspec/jbb/Company;SII)V(Unknown Source)
    at spec/jbb/JBBmain.runWarehouse(IIF)Z(Unknown Source)
    at spec/jbb/JBBmain.doIt()V(Unknown Source)
    at spec/jbb/JBBmain.main([Ljava/lang/String;)V(Unknown Source)
    at jrockit/vm/RNI.c2java(IIII)V(Native Method)
    -- end of trace

"Finalizer" id=8 idx=0x30 tid=4364 prio=8 alive, in native, native_waiting, daemon
    at jrockit/memory/Finalizer.getPendingFinalizee(Z)Ljava/lang/Object;(Native Method)
    at jrockit/memory/Finalizer.access$100(Finalizer.java:12)
    at jrockit/memory/Finalizer$4.run(Finalizer.java:131)
    at java/lang/Thread.run(Thread.java:619)
    at jrockit/vm/RNI.c2java(IIII)V(Native Method)
    -- end of trace
```

Using the Latency Tabs

Finding performance bottlenecks within your Java application is a bit of a detective's work. You know what the symptoms of the problem are, for example, the application is running really slow but the CPU isn't saturated. Where to start looking for clues to such an issue is tricky since most profiling tools for Java applications only pinpoint where in the code your application is spending the most time to run (which is a good start). What these tools tend to miss, however, or not show at all is where in the application stops and waits occur, i.e. where the application spends time being idle.

These stops and waits can be caused by poor memory management, such as limited heap space or a poorly managed heap that requires too many garbage collections. On the other hand the stops and waits can be latencies caused by multi-threaded applications that spend much of the processor time waiting, blocking, or sleeping. These problems have previously been hard to detect but now the JRA system is able to record latencies within your application and visualize running threads with their events in an easy to understand manner.

This section of the help gives you an overview of how you can use the latency tabs in the JRA Tool to work you way down to a Java application latency. In addition, you get one example of how a Java application that contains latencies looks on the **Latency Graph** tab and you will get an example workflow of how to use all latency tabs together. All in all, you now have a greater possibility to pinpoint where in the code waits and other latencies occur with the JRA latency capabilities.

This section is divided into the following topics:

- Latency Tabs at a Glance

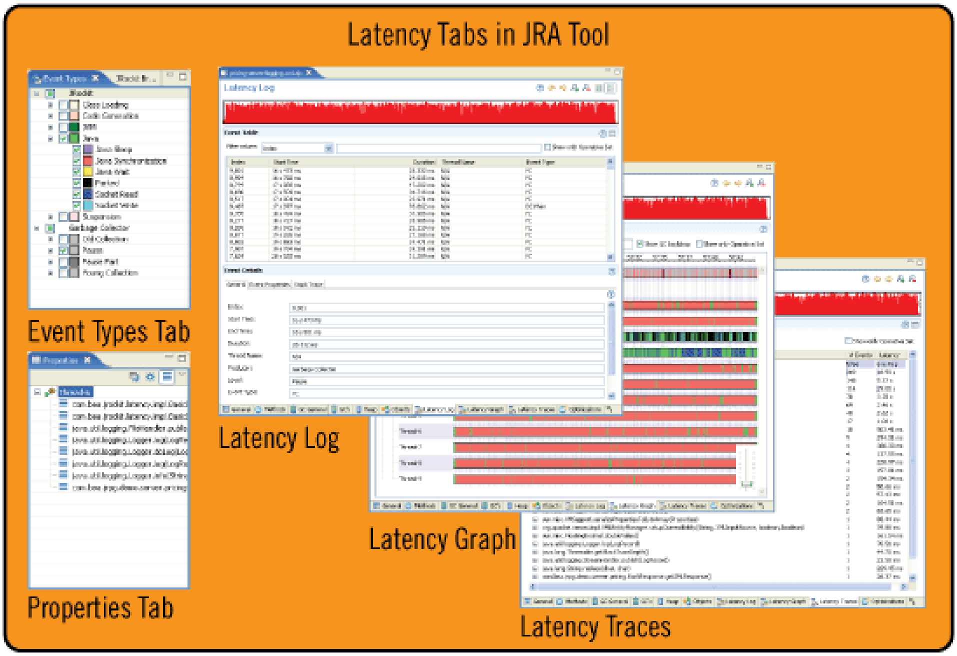
- Creating a JRA Recording with Latency Data
- Opening a JRA Recording that Contains Latency Data
- Shared Functionality Amongst All Latency Tabs
- Using the Latency Timeline Slide Bar
- What is an Operative Set?
- Working with an Operative Set
- About the Event Types Tab
- Using the Event Types Tab to Decrease Displayed Events
- Using the Event Types Tab to Work with Operative Sets
- About the Properties Tab
- Example of How to Compare two JRA Recordings where one Contains Latencies
- Example Workflow of How to Find Latencies

Latency Tabs at a Glance

The JRA Tool contains three tabs that all show latency data from different perspectives. These tabs are prefixed *Latency* and named: *Latency Log*, *Latency Graph*, and *Latency Traces* (Figure 16-1). Together with these three tabs, there two auxiliary tabs that allow you to turn on and off event types on the latency tabs and view properties.

Note: Depending on your settings in the Preferences (see Turning on/off Tabs), the latency tabs may be hidden when your recording does not contain latency information. See Creating a JRA Recording with Latency Data for information on how to enable latency information in your recordings.

Figure 16-1 Latency tabs at a glance



Creating a JRA Recording with Latency Data

You create a JRA Recording with latency data in pretty much the same way you create a regular JRA recording (see 3. Create JRA Recording). The difference, though, is that you use a different profile for creating a recording with latency data than when you create the "normal" JRA recording. If you use the profile with minimal overhead, the JRA will not perform a garbage collection at the end and beginning of the recording, which minimizes the impact on the system when creating a recording.

For help on the *Advanced* option for the recording profile, see To use the normal recording profile with advanced options.

The instructions for how to use the *Latency Recording* profiles are described in:

- To use the normal latency data profile
- To use the minimal latency recording profile

To use the normal latency data profile

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

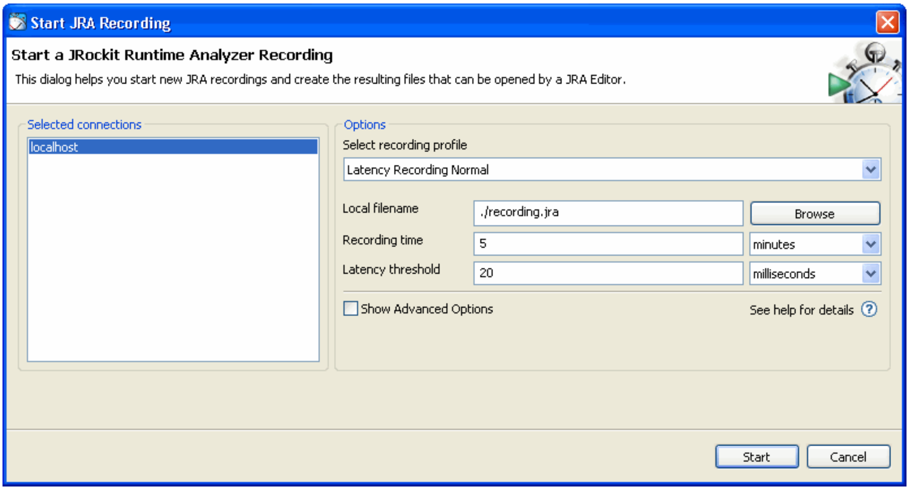
   If you run the application without load, the data captured from that application will not show where there is room for improvement.

2. In the **JRockit Browser**, select the JRockit instance you just started or select an entire folder with running JRockit instances.

3. Click the **Start JRA recording** button.

4. The **Start JRA Recording** dialog box appears (Figure 16-2).

Figure 16-2 JRA recording with normal latency profile



Start JRA Recording

Start a JRockit Runtime Analyzer Recording

This dialog helps you start new JRA recordings and create the resulting files that can be opened by a JRA Editor.

Selected connections: localhost

Options

Select recording profile: Latency Recording Normal

Local filename: ./recording.jra

Recording time: 5 minutes

Latency threshold: 20 milliseconds

☐ Show Advanced Options

5. Select the connection you want to record.

6. From the **Select recording file** drop-down list, choose **Latency Recording Normal**.

7. 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.

8. Set a recording time for the duration of the recording in the **Recording time** field.

9. Select the time unit you wish to use for specifying the recording time (minutes or seconds).

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.

10. Set a threshold value for **Latency threshold**. The latency threshold is the duration of the latency itself. As soon as the latency is longer than that threshold, the data will be saved.

    For advance option information, see To use the normal recording profile with advanced options.

11. Click **Start**.

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

To use the minimal latency recording profile

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

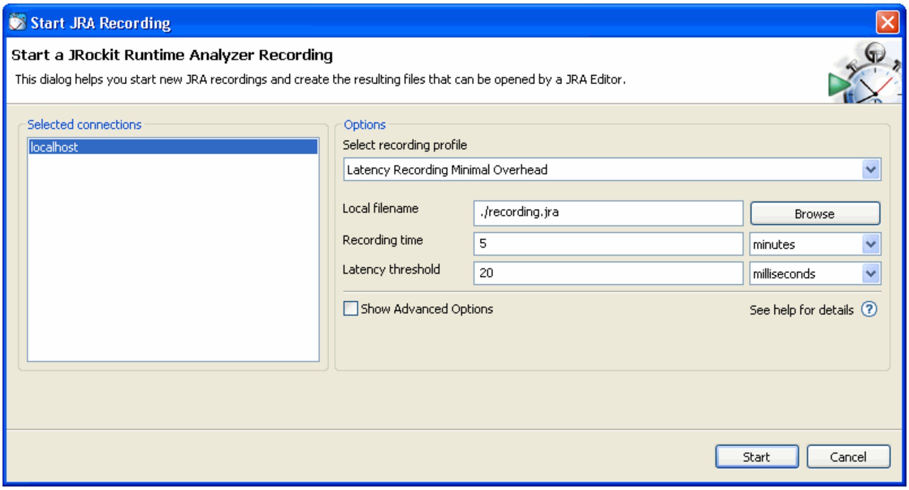
   If you run the application without load, the data captured from that application will not show where there is room for improvement.

2. In the **JRockit Browser**, select the JRockit instance you just started or select an entire folder with running JRockit instances.

3. Click the **Start JRA recording** button.

   The JRA Recording dialog box appears (Figure 16-3).

Figure 16-3 JRA recording with minimal latency overhead profile



Start JRA Recording

Start a JRockit Runtime Analyzer Recording

This dialog helps you start new JRA recordings and create the resulting files that can be opened by a JRA Editor.

Selected connections: localhost

Options

Select recording profile: Latency Recording Minimal Overhead

Local filename: ./recording.jra

Recording time: 5 minutes

Latency threshold: 20 milliseconds

☐ Show Advanced Options

4. Select the connection you want to record.
5. From the **Select recording file** drop-down list, choose **Latency Recording Minimal Overhead**.

   Minimal overhead means that the capturing of latency data affects the system in the least possible way, i.e. it will not disturb or put extra load when recording.

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 recording time for the duration of the recording in the **Recording time** field.
8. Select the time unit you wish to use for specifying the recording time (minutes or seconds).

   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.

9. Set a threshold value for **Latency threshold**. The latency threshold is the duration of the latency itself. As soon as the latency is longer than that threshold, the data will be saved.

   For advance option information, see To use the normal recording profile with advanced options.

10. Click **Start**.

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

Opening a JRA Recording that Contains Latency Data

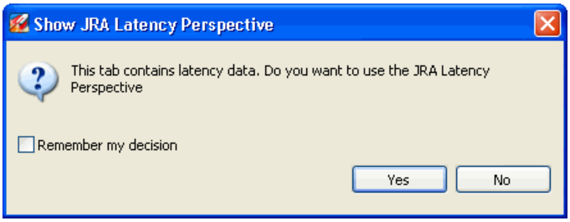
When JRockit is done recording a JRA file with latency data, the recording is automatically loaded in the JRA Tool.

To enable latency data on a latency tab

1. Click on a tab with the prefix *Latency*.

   The **Show JRA Latency Perspective** window opens (Figure 16-4).

Figure 16-4 The Show JRA Latency Perspective window



This tab contains latency data. Do you want to use the JRA Latency Perspective

☐ Remember my decision

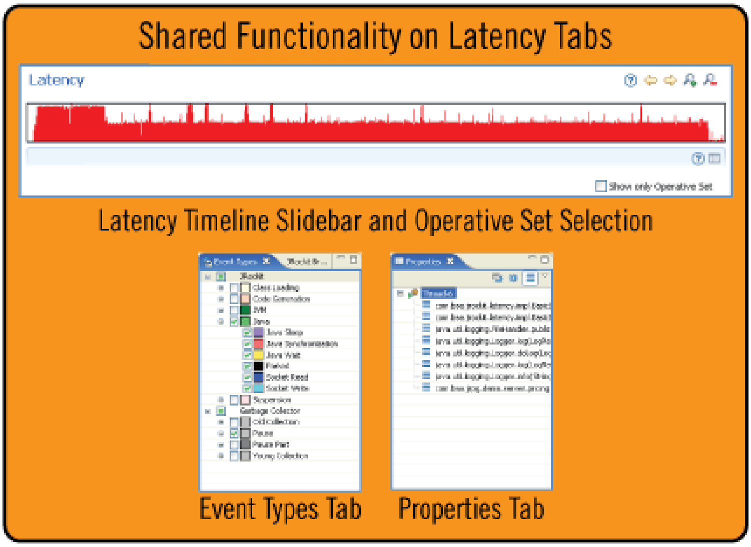
2. Select **Remember my selection** if you do not want this window to open the next time you click on a latency tab.

3. Click **Yes**.

Shared Functionality Amongst All Latency Tabs

The latency tabs have some functionality that they share, such as the Latency Timeline slide bar, the possibility to select events for the operative set, the Event Types tab, and the Properties tab (Figure 16-5).

Figure 16-5 Shared latency tab functionality


Shared Functionality on Latency Tabs
Latency
☐ Show only Operative Set
Latency Timeline Slidebar and Operative Set Selection
Event Types
Properties
Event Types Tab
Properties Tab

These topics explain the shared functionality of the latency tabs:

- Using the Latency Timeline Slide Bar
- What is an Operative Set?
- Working with an Operative Set
- About the Event Types Tab
- Using the Event Types Tab to Decrease Displayed Events
- About the Properties Tab

Using the Latency Timeline Slide Bar

The **Latency Timeline** slide bar is a universal slide bar for all tabs prefixed *Latency*. It shows the entire length of the recorded JRA file. Changing the time span or refocusing the Latency Timeline slide bar affects all latency tabs in the JRA Tool. You can also use the scroll and zoom buttons to refocus on events within the recording.

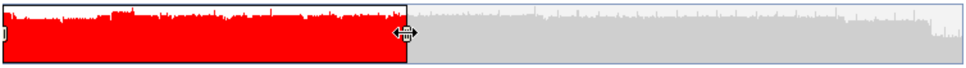
The different ways to use the Latency Timeline are described in the following topics:

- To decrease the time span on a latency tab
- To refocus using the timeline slide bar
- To move and zoom using the move and zoom buttons
- To reposition the timeline slide bar

To decrease the time span on a latency tab

- Click and drag the handles on the sides of the **Latency Timeline** (Figure 16-6).

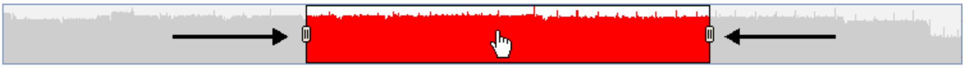
Figure 16-6 The Latency Timeline decreased



To refocus using the timeline slide bar

- Drag the **Latency Timeline** into the desired position (Figure 16-7).

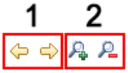
Figure 16-7 Refocus on the Latency Graph tab



To move and zoom using the move and zoom buttons

1. Click the move buttons (left or right) to move the Latency Timeline. The scroll buttons are marked 1 in Figure 16-8.

Figure 16-8 Move and zoom buttons



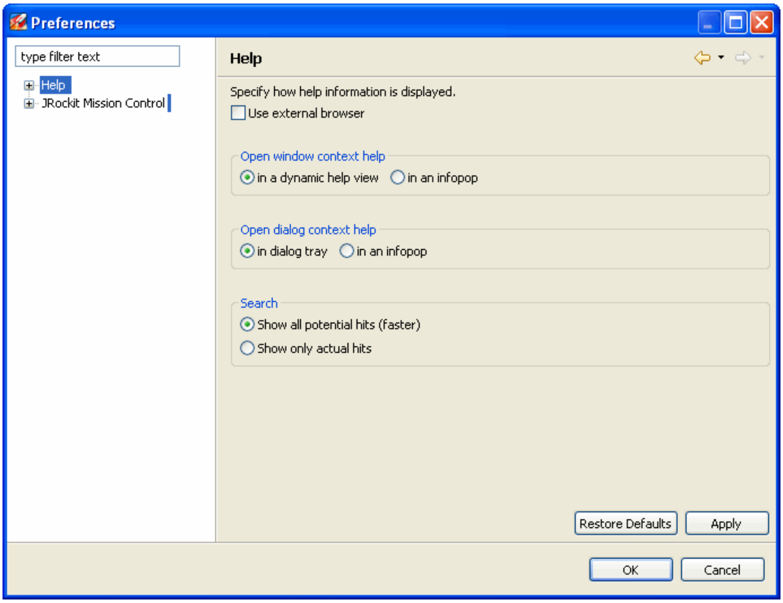
2. Click the zoom in/out buttons to shorten the time span on the Latency Timeline. The zoom in/out buttons are marked 2 in Figure 16-8.
3. Double-click the Latency Timeline slide bar to go back to display the full length of the latency data.

To reposition the timeline slide bar

1. On the Menu bar, select **Windows>Preferences...**

   The Preferences dialog box appears (Figure 16-9).

Figure 16-9 Preferences dialog box



Help

Specify how help information is displayed.

☐ Use external browser

Open window context help

● in a dynamic help view ○ in an infopop

Open dialog context help

● in dialog tray ○ in an infopop

Search

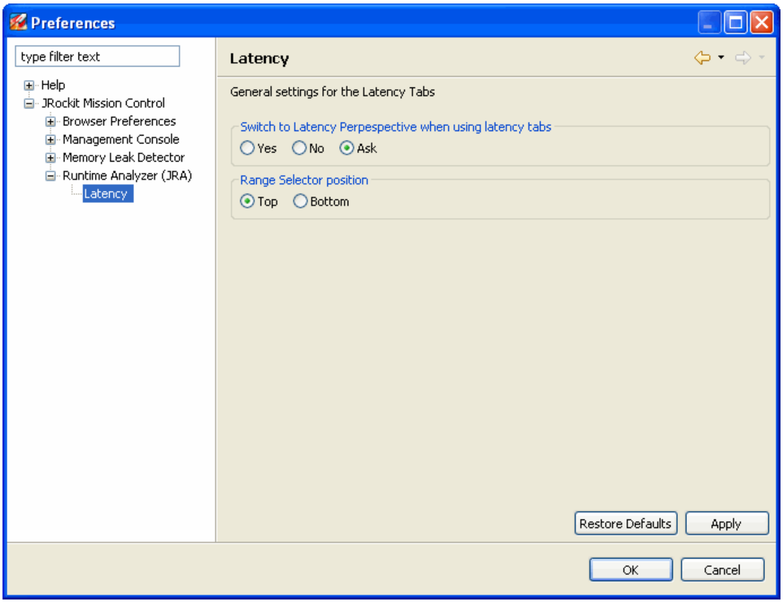
● Show all potential hits (faster)

○ Show only actual hits

2. In the left panel, select **JRockit Mission Control>Runtime Analyzer>Latency**.

   The L:atency panel appears (Figure 16-10).

Figure 16-10 Preferences dialog box—Latency panel



General settings for the Latency Tabs

Switch to Latency Perpespective when using latency tabs
○ Yes ○ No ● Ask

Range Selector position
● Top ○ Bottom

3. In the **Range Selector position** box, select the radio button that identifies where you want the timeline slide bar to appear; for example, if you want move the sliude bar to the bottom of the tab, select **Bottom**.

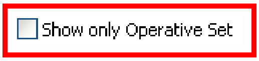
4. Select either **Apply** (if you want to set more preferences) or **OK**.

Note: For this change to take affect, you will need to close and reopen the particular recording.

What is an Operative Set?

An operative set is a set of events that you choose to work with. You can think of the operative set as a selection of events that you find particularly interesting to view. If you select events for the operative set on one latency tab, those events are remembered for the other latency tabs and you can easily view those events on the specific latency tab by selecting the **Show only Operative Set** option (Figure 16-11).

Figure 16-11 Show only Operative Set option



☐ Show only Operative Set

Working with an Operative Set

You can add and delete events in your operative set in different ways depending on which latency tab you are looking at at the moment. The procedures

- To select events for the operative set

  and

- To remove events for the operative set

describe how to add and remove events from the operative set. To look at the operative set, describes how to use the operative set within a tab. The ways on how to work with the operative set is similar on all latency tabs. These instructions describe how the operative set works for the Latency Log tab.

For an explanation of an operative set, see What is an Operative Set?.

To select events for the operative set

1. Click on any of the latency tabs, for example the Latency Log tab.
2. Right-click on one or select several events in the Event Table.
3. Select **Operative Set** > **Add selection** or **Operative Set** > **Set selection**.

   The **Add selection** option adds the events to an already existing operative set (or to a new one).

   The **Set selection** option clears and overrides the current operative set with the events that you currently have selected.

To remove events for the operative set

1. Click on any of the latency tabs, for example the Latency Log tab.
2. Right-click on one or select several events in the Event Table.
3. Select **Operative Set** > **Remove selection** or **Operative Set** > **Clear**.

   The **Remove selection** option deletes the currently selected events from an already existing operative set.

The **Clear** option deletes all events from an already existing operative set.

To look at the operative set

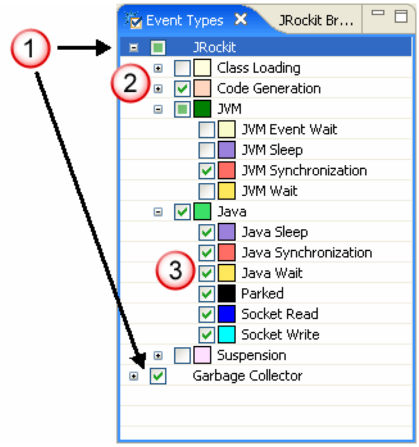
- Select the Show only Operative Set on the latency tab that you are at.

Notice that the list of events becomes more manageable.

About the Event Types Tab

The **Event Types** tab lists the events in relation to where they come from. The *Event types* themselves (marked 3 in Figure 16-12) come from a *Level* (marked 2 in Figure 16-12), and the Level comes from a *Producer* (marked 1 in Figure 16-12).

Figure 16-12 Producers, levels, and event types



- ▣ JRockit
  - ☐ Class Loading
  - ☑ Code Generation
  - ▣ JVM
    - ☐ JVM Event Wait
    - ☐ JVM Sleep
    - ☑ JVM Synchronization
    - ☐ JVM Wait
  - ☑ Java
    - ☑ Java Sleep
    - ☑ Java Synchronization
    - ☑ Java Wait
    - ☑ Parked
    - ☑ Socket Read
    - ☑ Socket Write
  - ☐ Suspension
- ☑ Garbage Collector

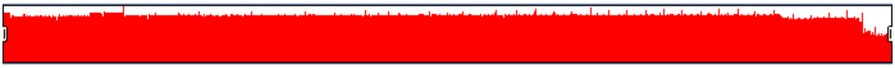
Below is an explanation of what you can see in the **Events Type** tab (Figure 16-12):

1. **Producers** are the part of the system that produced the events, for example, Garbage Collector and JRockit. A producer can come from a third party that uses the latency recording API.
2. **Levels** are a subdivision of producers. Two events within the same level and thread can never be performed at the same time. Levels are best visualized when looking at a thread in the Latency Graph tab (see Understanding the Different Parts of a Thread Image). There you see that the thread is divided into several levels, but two events within a level overlap.
3. **Event type** is the actual type of event that was responsible for the latency.

Using the Event Types Tab to Decrease Displayed Events

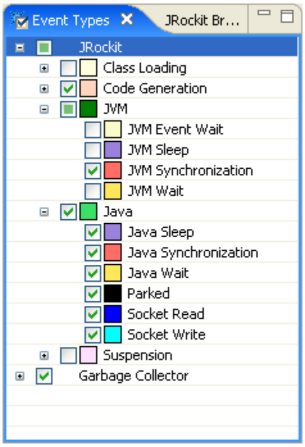
If you have many events selected in the **Event Types** tab, the Latency Timeline becomes quite saturated with information (Figure 16-13). It is a good idea to decrease the amount of events to eliminate events that are not interesting to view, for example, events that come from the JVM level.

Figure 16-13 Latency Timeline saturated with information



You decrease (or increase) the amount of data displayed in the Latency Timeline by deselecting events in the **Event Types** tab (Figure 16-14).

Figure 16-14 The Event Types tab



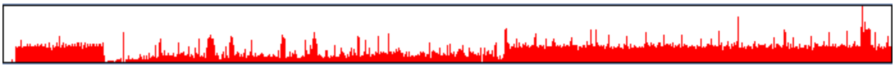
- ■ JRockit
  - ☐ Class Loading
  - ☑ Code Generation
  - ■ JVM
    - ☐ JVM Event Wait
    - ☐ JVM Sleep
    - ☑ JVM Synchronization
    - ☐ JVM Wait
  - ☑ Java
    - ☑ Java Sleep
    - ☑ Java Synchronization
    - ☑ Java Wait
    - ☑ Parked
    - ☑ Socket Read
    - ☑ Socket Write
  - ☐ Suspension
- ☑ Garbage Collector

To change the amount of events displayed

1. Click on the **Event Types** tab.
2. Click on a specific event, a level, or a producer to select or deselect (see Using the Event Types Tab to Decrease Displayed Events for an explanation of producer, level, and event).

The Latency Timeline in Figure 16-14 now looks something like Figure 16-15 when many of the events have been removed. Notice how much easier it is to see differences over time.

Figure 16-15 The Latency Timeline with events removed

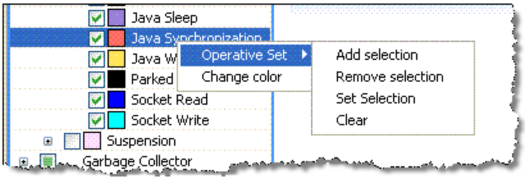


Using the Event Types Tab to Work with Operative Sets

Read about operative sets at What is an Operative Set?.

You can add events to an operative set or remove it from the set directly from the Event Types tab. This feature is useful when you want to add or remove all events of that type, add or remove all events from that specific thread, and so on. These features are enabled by using a context menu accessible from the Event Types tab (Figure 16-16).

Figure 16-16 Event Types operative set context menu



The procedures in this topic show you how:

- To select events for the operative set
- To remove events for the operative set

To select events for the operative set

1. Open the Event Types tab by opening the Windows menu and selecting **Show View** > **Operative Set**.
2. Right-click a type in the Event Types tab (see About the Event Types Tab for a description of tab contents).

   The context menu appears.
3. Select **Operative Set** > **Add selection** or **Operative Set** > **Set selection**.

   The **Add selection** option adds the type to an already existing operative set (or to a new one). Adding the event type to operative set adds all events of the selected type to the set.

   The **Set selection** option clears and overrides the current operative set with the types that you currently have selected.

To remove events for the operative set

1. Open the Event Types tab by opening the Windows menu and selecting **Show View** > **Operative Set**.
2. Right-click an event in the Event Types tab.

   The context menu appears.
3. Select **Operative Set** > **Remove selection** or **Operative Set** > **Clear**.

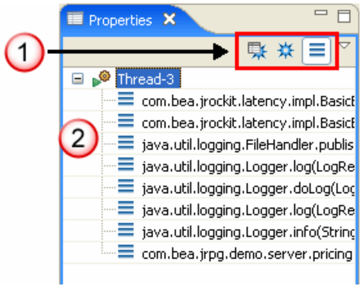
   The **Remove selection** option deletes the currently selected events from an already existing operative set.

   The **Clear** option deletes all events from an already existing operative set.

About the Properties Tab

The **Properties** tab lists the event properties, the event's stack trace, or the general event data depending on the view you have chosen (Figure 16-17). You select view by clicking on the button that corresponds to the view you want to see.

Figure 16-17 Properties tab



```
Thread-3
    com.bea.jrockit.latency.impl.Basic
    com.bea.jrockit.latency.impl.Basic
    java.util.logging.FileHandler.publis
    java.util.logging.Logger.log(LogRe
    java.util.logging.Logger.doLog(Lo
    java.util.logging.Logger.log(LogRe
    java.util.logging.Logger.info(Strin
    com.bea.jrpg.demo.server.pricing
```

Below is an explanation of what you can see on the **Properties** tab (Figure 16-17):

1. Buttons for choosing the property you want to view.

Table 16-1 gives an explanation to the different buttons.

Table 16-1 Properties tab buttons

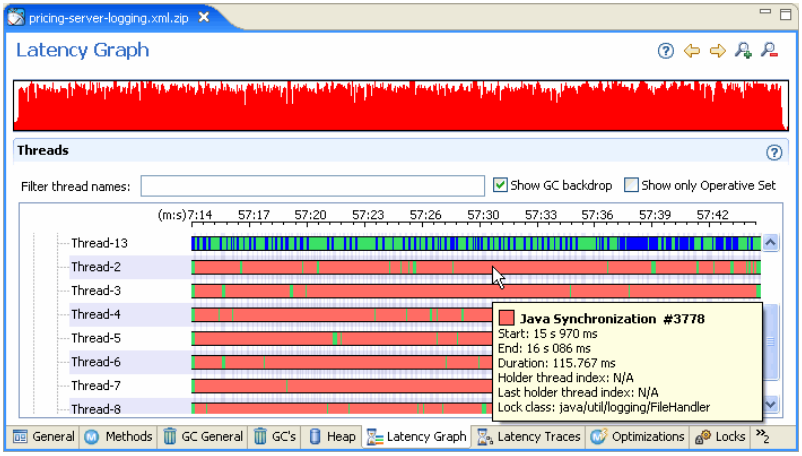
Button	Description
▤	**Event Properties** button. Shows the properties of the specific event. These properties are the same as found on the Event Details panel on the **Latency Log** tab.
✹	**Event General** button. Shows keys and their respective values for each event.
☰	**Stack Trace** button. Shows the stack trace for a specific event.

2. List of information. This list changes content depending on the button you click in the tab.

Example of How to Compare two JRA Recordings where one Contains Latencies

In this example you will see two recordings from the same application. The application that has been recorded uses a common method for logging transactions, which causes many latencies due to Java synchronization. These latencies can be found in almost all threads in the recording that is named *pricing-server-logging.xml.zip* (Figure 16-18).

Figure 16-18 pricing-serving-logging.xml.zip with latencies



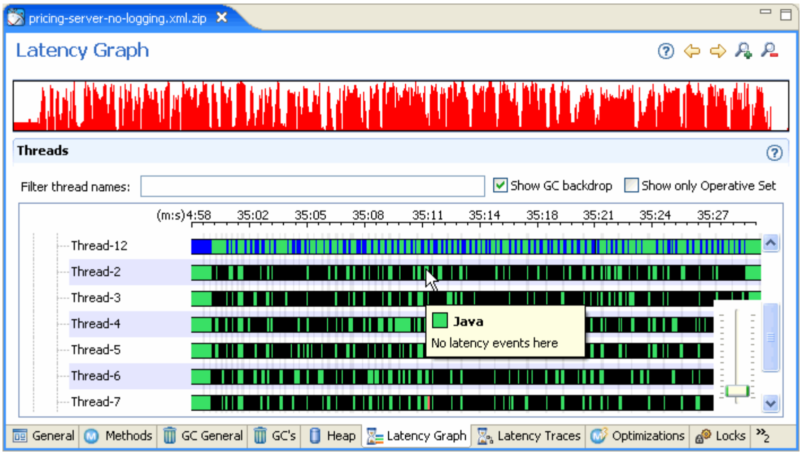
Filter thread names:

☑ Show GC backdrop ☐ Show only Operative Set

🟥 **Java Synchronization #3778**
Start: 15 s 970 ms
End: 16 s 086 ms
Duration: 115.767 ms
Holder thread index: N/A
Last holder thread index: N/A
Lock class: java/util/logging/FileHandler

For the second recording the same application has been used, but the calls to the logging system has been removed, which causes a lot less latencies in the system. The second recording is named *pricing-server-no-logging.xml.zip* (Figure 16-19). You see the difference both in the color scheme and the Latency Timeline slide bar.

Figure 16-19 pricing-server-no-logging.xml.zip with no latencies



Filter thread names:

☑ Show GC backdrop ☐ Show only Operative Set

🟩 **Java**
No latency events here

You can compare the two JRA recordings next to each other within the JRA Tool, which makes it easier to see what has happened with the changes in the application (see To compare and contrast JRA recordings for information on how to compare recordings).

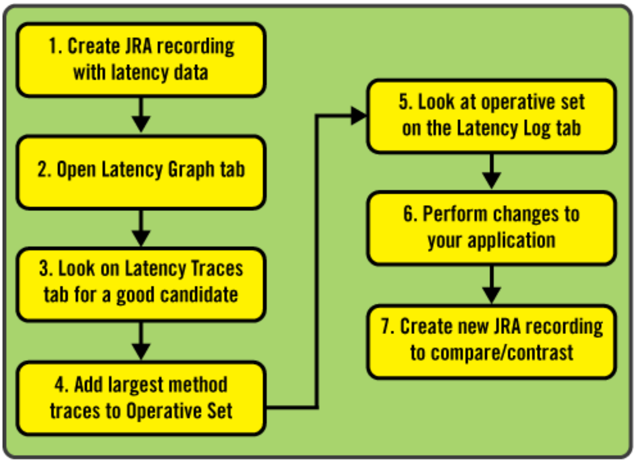
Example Workflow of How to Find Latencies

The application that has been used in this example contains a common method for logging transactions, which causes many latencies due to Java synchronization. These latencies can be found in almost all threads in the recording. This section will guide you through how the JRA Tool can be used to find which method that contains the latency.

Note: This is an example recording that contains extremely visible latencies, the application that you are looking at might not contain as obvious latencies.

Look in Figure 16-20 for an example workflow of how to start your latency detective work.

Figure 16-20 Example workflow for finding latencies


1. Create JRA recording with latency data
2. Open Latency Graph tab
3. Look on Latency Traces tab for a good candidate
4. Add largest method traces to Operative Set
5. Look at operative set on the Latency Log tab
6. Perform changes to your application
7. Create new JRA recording to compare/contrast

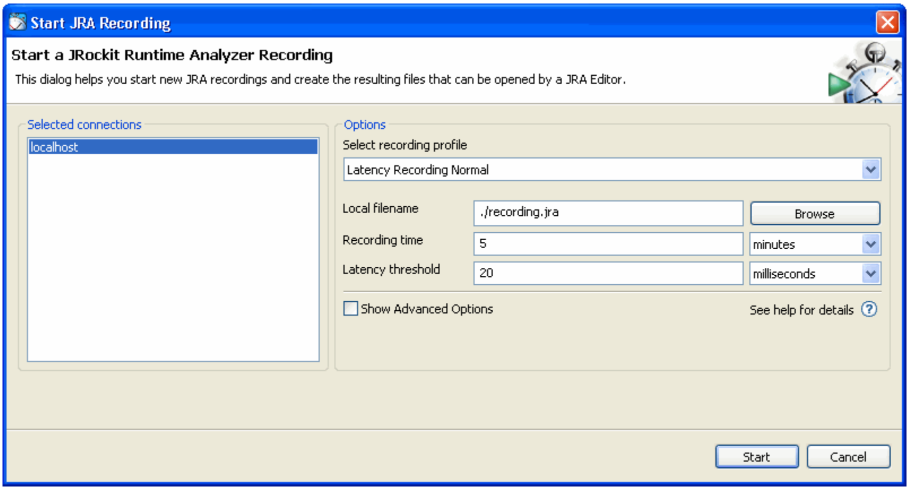
The workflow is divided into the following instructions:

- 1. Create a JRA Recording with Latency Data
- 2. Open the JRA Recording in the Latency Graph Tab
- 3. Look on the Latency Traces Tab to Find Specific Method
- 4. Add a Suspected Method to the Operative Set
- 5. Look at Operative Set on the Latency Traces Tab
- 6. Perform Changes to Your Application
- 7. Compare and Contrast Recordings

1. Create a JRA Recording with Latency Data

Before you start profiling your Java application, you need to create a JRA Recording with the latency recording profile (Figure 16-21). See Creating a JRA Recording with Latency Data for instructions on how to create a recording.

Figure 16-21 JRA recording with normal latency profile



Start JRA Recording

Start a JRockit Runtime Analyzer Recording

This dialog helps you start new JRA recordings and create the resulting files that can be opened by a JRA Editor.

Selected connections: localhost

Options

Select recording profile: Latency Recording Normal

Local filename: ./recording.jra

Recording time: 5 minutes

Latency threshold: 20 milliseconds

☐ Show Advanced Options

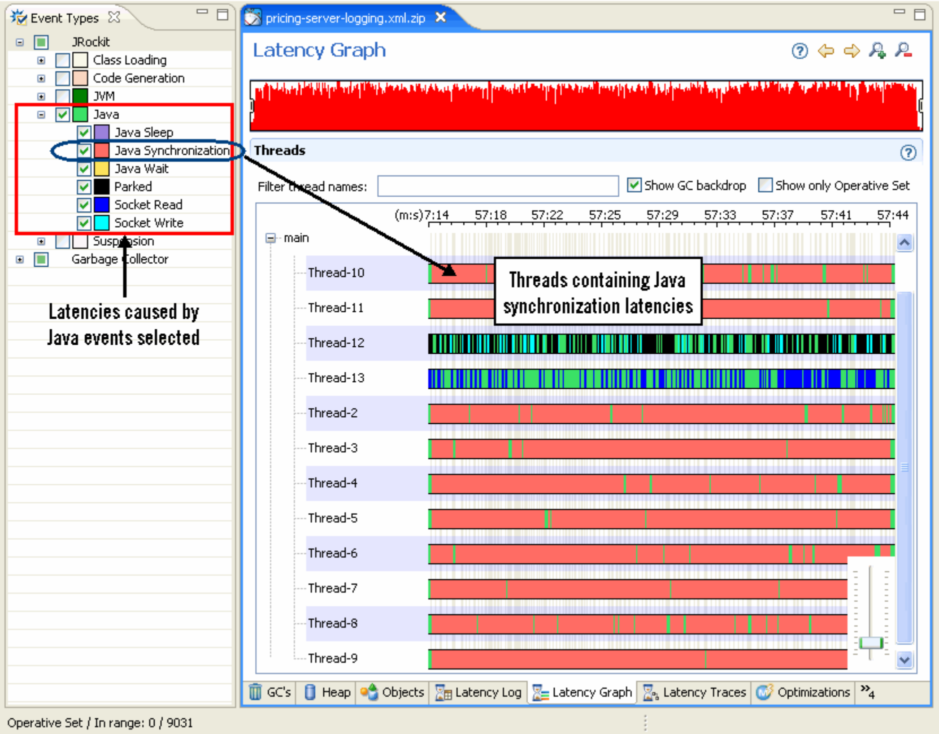
See help for details

Investigate further by opening the JRA recording, see 2. Open the JRA Recording in the Latency Graph Tab.

2. Open the JRA Recording in the Latency Graph Tab

Open your JRA recording that contains latency data and click on the Latency Graph tab to see an overview of all threads. This tab offers a great overview of a first glance to find latencies. In Figure 16-22, the Latency Graph tab is visible and possible latency events from the Java producer has been selected in the Event Types tab. The color of the Java Synchronization event in almost is visible in almost all threads of the application, which gives a hint that the Java Synchronization event causes latencies. Investigate further by looking at the Latency Traces tab, see 3. Look on the Latency Traces Tab to Find Specific Method.

Figure 16-22 Latency Graph tab with Java events selected



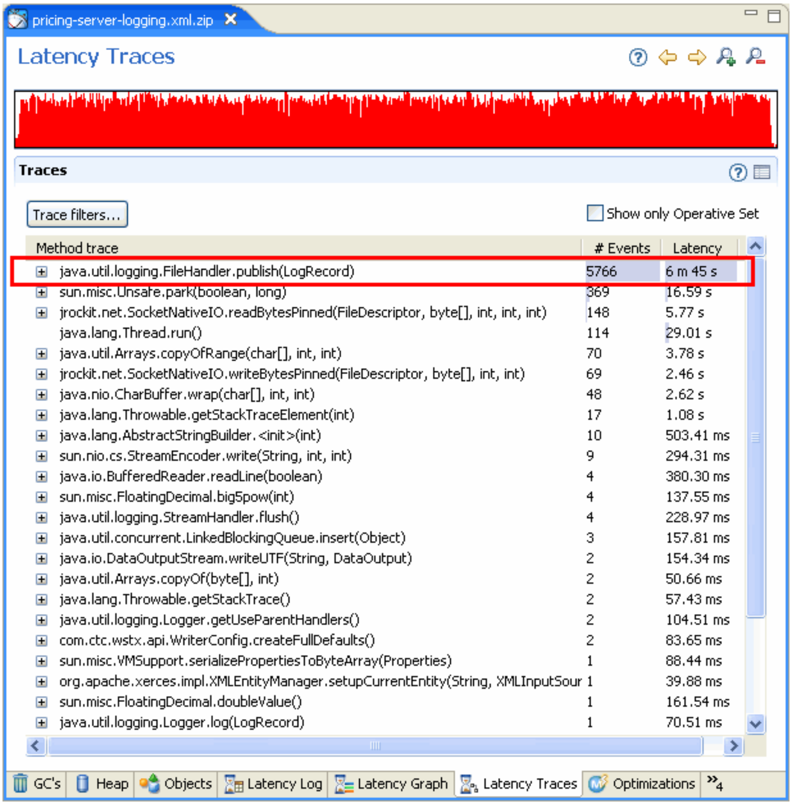
Latencies caused by Java events selected

Threads containing Java synchronization latencies

3. Look on the Latency Traces Tab to Find Specific Method

Once you are done viewing your recording from a threads perspective, you click on the Latency Traces tab to find methods that contain latencies. The Traces table is sorted to show the methods that contain the most number of events with latencies first. Figure 16-23 shows that the most latencies are within the method `java.util.logging.FileHandler.publish(LogRecord)`. Investigate further by adding the method `java.util.logging.FileHandler.publish(LogRecord)` to the operative set, see 4. Add a Suspected Method to the Operative Set.

Figure 16-23 Latency Traces tab with method that contains latencies



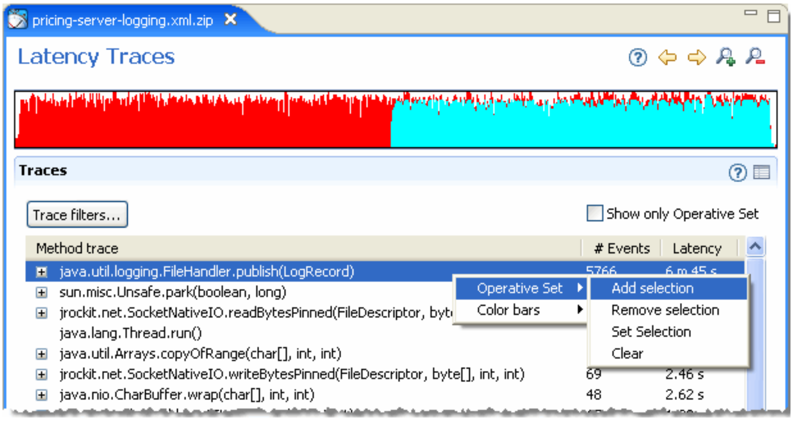
Method trace	# Events	Latency
java.util.logging.FileHandler.publish(LogRecord)	5766	6 m 45 s
sun.misc.Unsafe.park(boolean, long)	369	16.59 s
jrockit.net.SocketNativeIO.readBytesPinned(FileDescriptor, byte[], int, int, int)	148	5.77 s
java.lang.Thread.run()	114	29.01 s
java.util.Arrays.copyOfRange(char[], int, int)	70	3.78 s
jrockit.net.SocketNativeIO.writeBytesPinned(FileDescriptor, byte[], int, int)	69	2.46 s
java.nio.CharBuffer.wrap(char[], int, int)	48	2.62 s
java.lang.Throwable.getStackTraceElement(int)	17	1.08 s
java.lang.AbstractStringBuilder.<init>(int)	10	503.41 ms
sun.nio.cs.StreamEncoder.write(String, int, int)	9	294.31 ms
java.io.BufferedReader.readLine(boolean)	4	380.30 ms
sun.misc.FloatingDecimal.big5pow(int)	4	137.55 ms
java.util.logging.StreamHandler.flush()	4	228.97 ms
java.util.concurrent.LinkedBlockingQueue.insert(Object)	3	157.81 ms
java.io.DataOutputStream.writeUTF(String, DataOutput)	2	154.34 ms
java.util.Arrays.copyOf(byte[], int)	2	50.66 ms
java.lang.Throwable.getStackTrace()	2	57.43 ms
java.util.logging.Logger.getUseParentHandlers()	2	104.51 ms
com.ctc.wstx.api.WriterConfig.createFullDefaults()	2	83.65 ms
sun.misc.VMSupport.serializePropertiesToByteArray(Properties)	1	88.44 ms
org.apache.xerces.impl.XMLEntityManager.setupCurrentEntity(String, XMLInputSour	1	39.88 ms
sun.misc.FloatingDecimal.doubleValue()	1	161.54 ms
java.util.logging.Logger.log(LogRecord)	1	70.51 ms

4. Add a Suspected Method to the Operative Set

When you have found a method that contains latencies, you can add that to the operative set. By adding the method to the operative set, you can concentrate your viewing to the pieces of information that you are mostly interested in viewing even on other latency tabs. Figure 16-24 shows how to add the method `java.util.logging.FileHandler.publish(LogRecord)` to the operative set. Notice how the Latency Timeline changes color (the operative set becomes blue) when you have made a selection to the operative set.

Investigate further by looking at the method that you have selected to the operative set in the Latency Log tab, see 5. Look at Operative Set on the Latency Traces Tab.

Figure 16-24 Adding method to operative set



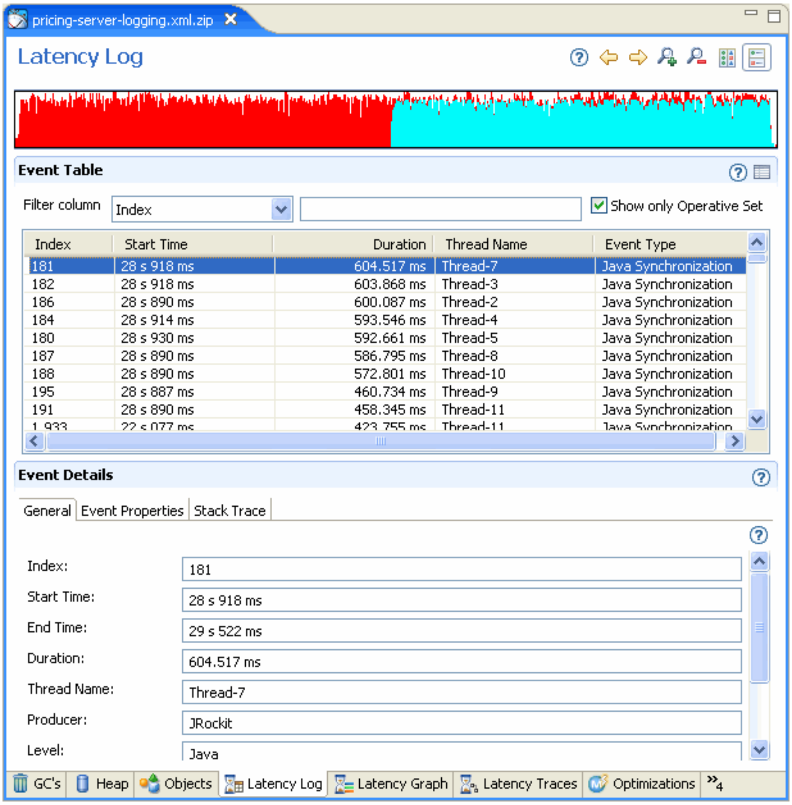
Method trace	# Events	Latency
java.util.logging.FileHandler.publish(LogRecord)	5766	6 m 45 s
sun.misc.Unsafe.park(boolean, long)		
jrockit.net.SocketNativeIO.readBytesPinned(FileDescriptor, byt		
java.lang.Thread.run()		
java.util.Arrays.copyOfRange(char[], int, int)		
jrockit.net.SocketNativeIO.writeBytesPinned(FileDescriptor, byte[], int, int)	69	2.46 s
java.nio.CharBuffer.wrap(char[], int, int)	48	2.62 s

5. Look at Operative Set on the Latency Traces Tab

The Latency Log tab presents, in a sorted list, events that contain the most latency. In Figure 16-25 only the operative set is shown and you see that the first event is causing latencies in Thread-7. Look at the Event Details panel for property and stack trace information.

Now you might have a pretty good idea of where in the code you need to perform changes. Perform those changes and create a new JRA recording to compare and contrast the results, see 6. Perform Changes to Your Application.

Figure 16-25 Looking at operative set on the Latency Log tab



Filter column: Index

☑ Show only Operative Set

Index	Start Time	Duration	Thread Name	Event Type
181	28 s 918 ms	604.517 ms	Thread-7	Java Synchronization
182	28 s 918 ms	603.868 ms	Thread-3	Java Synchronization
186	28 s 890 ms	600.087 ms	Thread-2	Java Synchronization
184	28 s 914 ms	593.546 ms	Thread-4	Java Synchronization
180	28 s 930 ms	592.661 ms	Thread-5	Java Synchronization
187	28 s 890 ms	586.795 ms	Thread-8	Java Synchronization
188	28 s 890 ms	572.801 ms	Thread-10	Java Synchronization
195	28 s 887 ms	460.734 ms	Thread-9	Java Synchronization
191	28 s 890 ms	458.345 ms	Thread-11	Java Synchronization
1 933	22 s 077 ms	423.755 ms	Thread-11	Java Synchronization

Index: 181

Start Time: 28 s 918 ms

End Time: 29 s 522 ms

Duration: 604.517 ms

Thread Name: Thread-7

Producer: JRockit

Level: Java

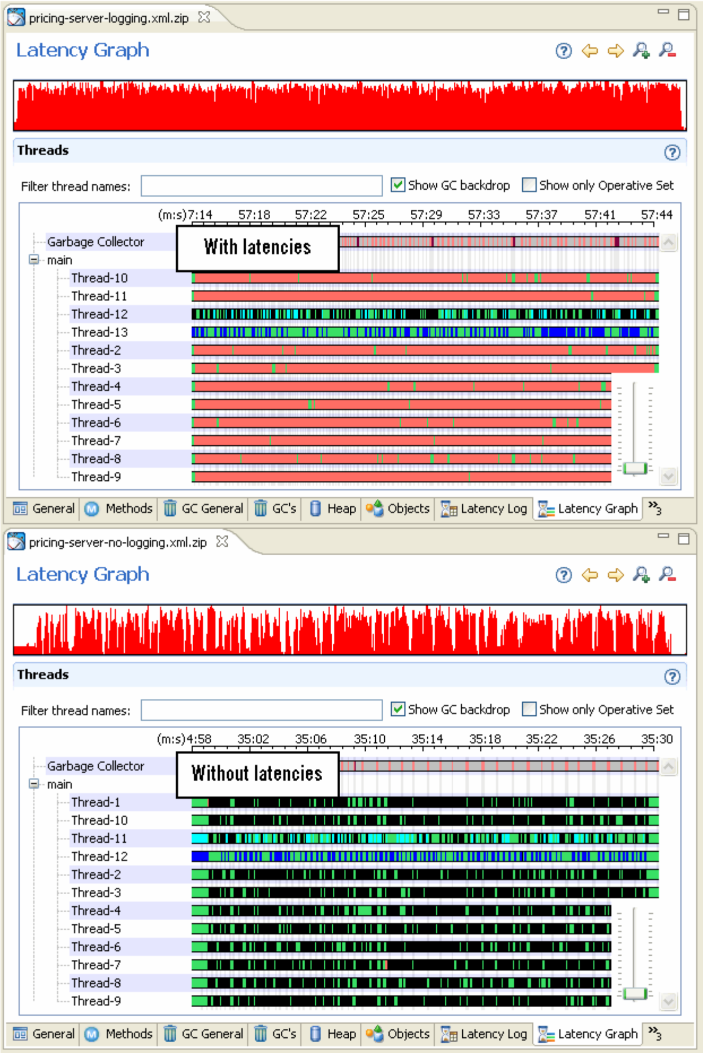
6. Perform Changes to Your Application

Once you have found which methods and events that cause latency problems you need to perform changes to your application code. Perform those changes, create a new JRA recording, and compare and contrast the result, see 7. Compare and Contrast Recordings.

7. Compare and Contrast Recordings

The latency problem has now been fixed in the example application and the result can look something similar to what you see in Figure 16-26.

Figure 16-26 Logging method reworked



Filter thread names: ☑ Show GC backdrop ☐ Show only Operative Set

Filter thread names: ☑ Show GC backdrop ☐ Show only Operative Set

Latency Log Information

The **Latency Log** tab lists the latency events that took place during the recording. By looking at latency data in the Latency Log tab, you can easily find a specific event type or select an attribute by using the sort and filter functions.

Note: The latency events that are recorded do not necessarily mean that they cause any problems in the running of the application.

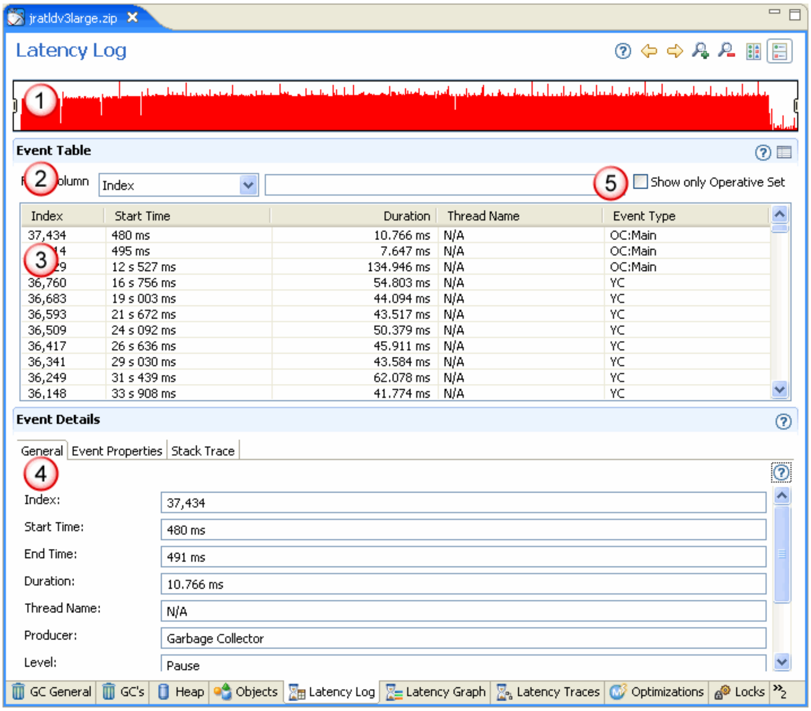
This section is divided into the following topics:

- Getting Familiar with the Latency Log Tab
- Changing Start Time View on an Event
- About Details for Events
- Selecting an Event
- Understanding Event Details
- Viewing General Event Details
- Viewing Event Property Details
- Viewing Event Stack Traces

Getting Familiar with the Latency Log Tab

The **Latency Log** tab contains Latency Timeline information, an Event Table, and Event Details (Figure 17-1).

Figure 17-1 The Latency Log tab



Filter column: Index

☐ Show only Operative Set

Index	Start Time	Duration	Thread Name	Event Type
37,434	480 ms	10.766 ms	N/A	OC:Main
14	495 ms	7.647 ms	N/A	OC:Main
29	12 s 527 ms	134.946 ms	N/A	OC:Main
36,760	16 s 756 ms	54.803 ms	N/A	YC
36,683	19 s 003 ms	44.094 ms	N/A	YC
36,593	21 s 672 ms	43.517 ms	N/A	YC
36,509	24 s 092 ms	50.379 ms	N/A	YC
36,417	26 s 636 ms	45.911 ms	N/A	YC
36,341	29 s 030 ms	43.584 ms	N/A	YC
36,249	31 s 439 ms	62.078 ms	N/A	YC
36,148	33 s 908 ms	41.774 ms	N/A	YC

Index: 37,434
Start Time: 480 ms
End Time: 491 ms
Duration: 10.766 ms
Thread Name: N/A
Producer: Garbage Collector
Level: Pause

The **Latency Log** tab is divided into the following sections:

1. **Latency Timeline** slide bar—this timeline shows the entire recording in its full length (the Latency Timeline works the same on all tabs that start with the name *Latency*, see Using the Latency Timeline Slide Bar for more information).
2. **Filter column**—see Filtering Information on how to use this function.
3. **Event Table**—the Event Table lists all events that took place during the recording.
4. **Event Details**—this panel lists the most common types on the heap at the end of the recording.
5. **Show only Operative Set**—this option allows you to concentrate on studying the events that you have chosen for your operative set (see What is an Operative Set? for a description of the operative set and how to use it).

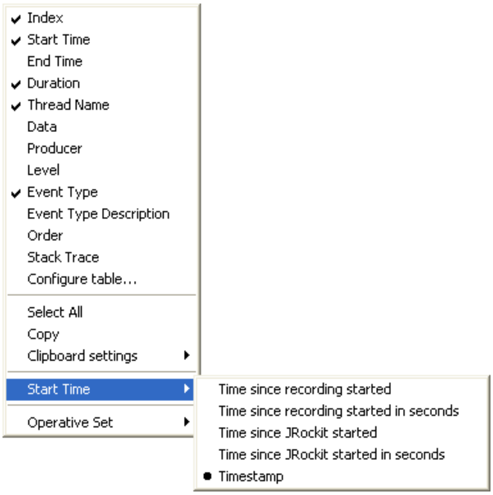
Changing Start Time View on an Event

The JRA recording collects data for different start times of a recording, for example, time since the recording started and since JRockit started. This section describes how to change the view of the start time.

To change the start time view

1. Right click the **Event Table** (or right click on the label **Start Time** on the **General** tab under the **Event Details** panel).
2. Click **Start Time**.

Figure 17-2 Start Time selection



3. Select one of the following **Start Time** views:
   - **Time since recording started:** this is the default view. It shows how much time has elapsed since the JRA recording started (in seconds and milliseconds).
   - **Time since recording started in seconds**: this is a shorter view where you see how many seconds have elapsed since the recording started.
   - **Time since JRockit started**: this view is useful when you have created a recording and used a JRockit that has, for example, been running on your network for a period of time. This viewing option shows the time divided in hours, minutes, and seconds.

  - **Time since JRockit started in seconds**: this view is useful when you have created a recording and used a JRockit that has, for example, been running on your network for a period of time. This viewing option shows the time in seconds.
  - **Timestamp**: this view shows actual time and date for when the event happened (on the computer that is running your application).

About Details for Events

The **Event Table** list on the **Latency Log** tab lists all latency events that have taken place during the recording provided the latency sampling was enabled during the recording (see 3. Create JRA Recording for information on how to record latency data).

- Selecting an Event
- Understanding Event Details

Selecting an Event

There are two places you can view the details for an event: on the **Latency Log** tab panel called **Event Details** or in the **Properties** tab.

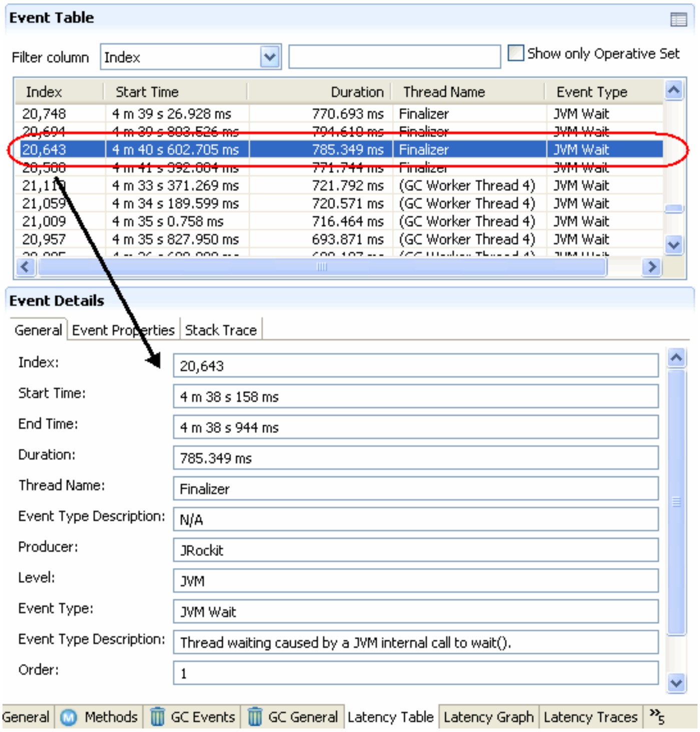
- To select an event and view its details under the Event Details
- About the Properties Tab

To select an event and view its details under the Event Details

1. Click the event for which you want to view details.

   The event specifics are listed in the panel called **Event Details** (Figure 17-3).

Figure 17-3 Event selected with General Event Details



Index	Start Time	Duration	Thread Name	Event Type
20,748	4 m 39 s 26.928 ms	770.693 ms	Finalizer	JVM Wait
20,694	4 m 39 s 803.526 ms	794.610 ms	Finalizer	JVM Wait
20,643	4 m 40 s 602.705 ms	785.349 ms	Finalizer	JVM Wait
20,588	4 m 41 s 392.004 ms	771.744 ms	Finalizer	JVM Wait
21,11[illegible]	4 m 33 s 371.269 ms	721.792 ms	(GC Worker Thread 4)	JVM Wait
21,059	4 m 34 s 189.599 ms	720.571 ms	(GC Worker Thread 4)	JVM Wait
21,009	4 m 35 s 0.758 ms	716.464 ms	(GC Worker Thread 4)	JVM Wait
20,957	4 m 35 s 827.950 ms	693.871 ms	(GC Worker Thread 4)	JVM Wait

Index: 20,643
Start Time: 4 m 38 s 158 ms
End Time: 4 m 38 s 944 ms
Duration: 785.349 ms
Thread Name: Finalizer
Event Type Description: N/A
Producer: JRockit
Level: JVM
Event Type: JVM Wait
Event Type Description: Thread waiting caused by a JVM internal call to wait().
Order: 1

You can also view the event specifics on the **Properties** tab

2. Click on the different tabs in the **Event Details** panel to see different aspects of detail for the event.

   Note: If you select several events, the **Event Details** tabs show the information for the event that was selected first.

Understanding Event Details

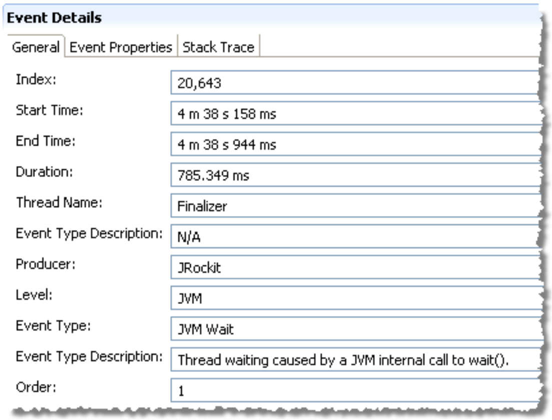
As described under Selecting an Event, you have two possibilities to view details of an event: either directly on the **Latency Log** tab or in the **Properties** tab. You also have the possibility to view the details next to each other, for example, view **General** event details on the **Properties** tab and **Stack Trace** details on the **Event Details** panel. Either way, the information is the same. The description in this help depicts how the event details are displayed from the **Latency Log** view. The **Event Details** are divided into the following sections:

- Viewing General Event Details
- Viewing Event Property Details
- Viewing Event Stack Traces

Viewing General Event Details

To view general event details either click the General tab under Event Details or click the General button on the Properties tab (Figure 17-4). The General events details is an overview of general specifics for the selected event (Figure 17-4).

Figure 17-4 The General tab for a thread event



Index:	20,643
Start Time:	4 m 38 s 158 ms
End Time:	4 m 38 s 944 ms
Duration:	785.349 ms
Thread Name:	Finalizer
Event Type Description:	N/A
Producer:	JRockit
Level:	JVM
Event Type:	JVM Wait
Event Type Description:	Thread waiting caused by a JVM internal call to wait().
Order:	1

To view General event details

1. Select an event in the **Event Table**.
2. Click the **General** tab in the **Event Details** panel (Figure 17-4).

   The following information can be found on the **General** tab. If something is marked N/A, it means that there was no information for that piece of information during the recording.

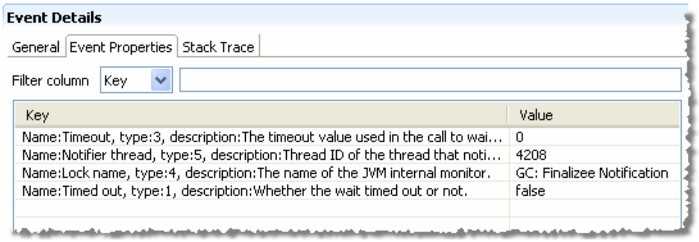
   - **Index**—a number that keeps track of each event in the recording.
   - **Start Time**—indicates the time from when the recording was started (default). You can change the default setting of the start time, see Changing Start Time View on an Event.
   - **End Time**—indicates when the specific event stopped.
   - **Duration**—the length of the event measured in milliseconds.

- **Thread Name**—the name of the thread you are inspecting.
- **Producer**—the part of the system that produced the thread, for example, Garbage Collector and JRockit.
- **Level**—Levels are a subdivision of producers. Two events within the same level and thread can never be performed at the same time. When you click on several event types within a level, the events appear on top of each other (see Using the Event Types Tab to Decrease Displayed Events).
- **Event Type**—a subdivision of levels. The Event Type corresponds to what you have selected on the **Event Types** tab (see Using the Event Types Tab to Decrease Displayed Events).
- **Event Type Description**—a brief description of the event type, for example, *Thread waiting for a JVM internal event.*

Viewing Event Property Details

Select an event in the **Event Table** and view its details on the tab **Event Properties** tab (Figure 17-5).

Figure 17-5 The Event Properties tab



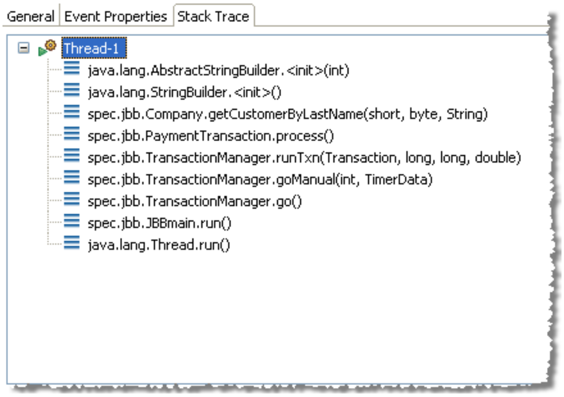
Key	Value
Name:Timeout, type:3, description:The timeout value used in the call to wai…	0
Name:Notifier thread, type:5, description:Thread ID of the thread that noti…	4208
Name:Lock name, type:4, description:The name of the JVM internal monitor.	GC: Finalizee Notification
Name:Timed out, type:1, description:Whether the wait timed out or not.	false

Use the filtering function to find, for example, a specific value, see Filtering Information.

Viewing Event Stack Traces

The **Stack Trace** tab shows all events on the stack that lead up to the event that you are currently monitoring (Figure 17-6).

Figure 17-6 The Stack Traces tab



```
Thread-1
    java.lang.AbstractStringBuilder.<init>(int)
    java.lang.StringBuilder.<init>()
    spec.jbb.Company.getCustomerByLastName(short, byte, String)
    spec.jbb.PaymentTransaction.process()
    spec.jbb.TransactionManager.runTxn(Transaction, long, long, double)
    spec.jbb.TransactionManager.goManual(int, TimerData)
    spec.jbb.TransactionManager.go()
    spec.jbb.JBBmain.run()
    java.lang.Thread.run()
```

To view the event stack trace

1. Select an event in the **Event Table**.
2. Click the **Stack Trace** tab in the **Event Details** panel (Figure 17-6).

CHAPTER 18

Latency Graph Information

The **Latency Graph** gives you a graphical overview of how the application executes and it is easy to select events in terms of when they happened and in which thread. You have a possibility to both zoom in on a shorter time interval and to magnify the threads themselves to better see the different events that occurred in the thread.

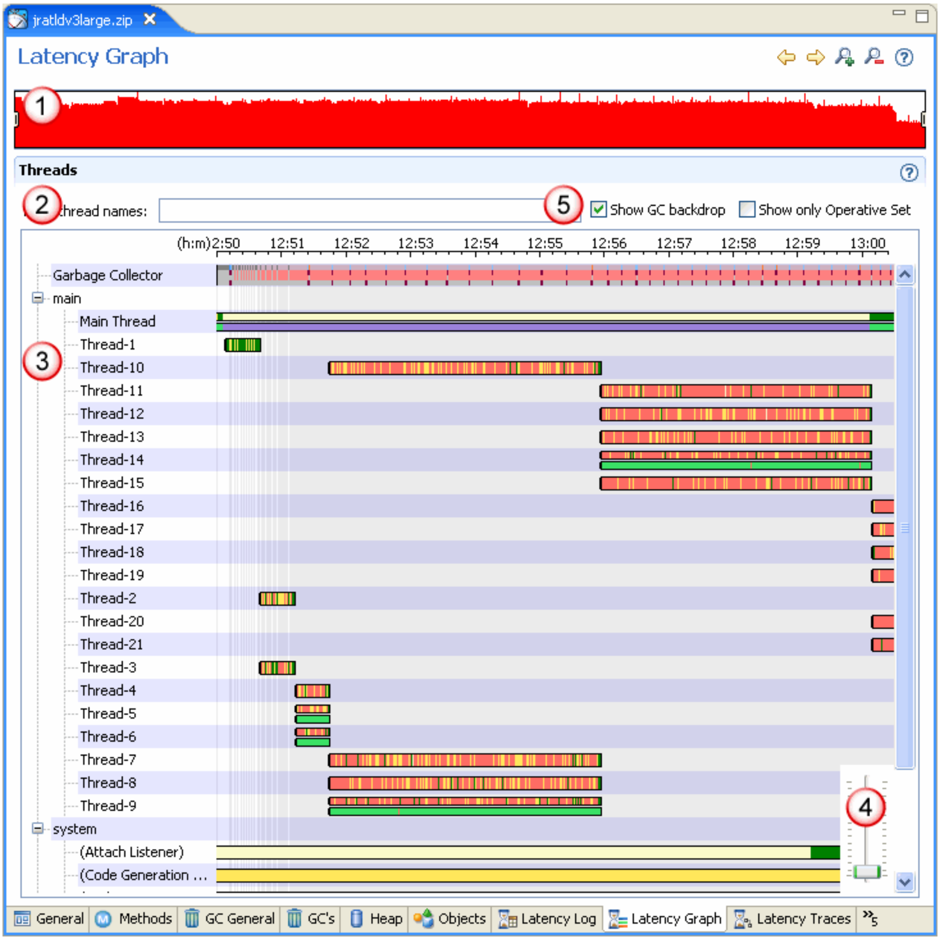
This section is divided into the following topics:

- Getting Familiar with the Latency Graph Tab
- Using the Latency Timeline Slide Bar
- Filtering on Thread Names
- What Does the Threads Chart Contain?
- Magnifying a Thread
- Showing Garbage Collection Backdrop

Getting Familiar with the Latency Graph Tab

The **Latency Graph** tab (Figure 18-1) displays the Latency Timeline and the Threads graph.

Figure 18-1 The Latency Graph tab



☑ Show GC backdrop ☐ Show only Operative Set

The **Latency Graph** tab is divided into the following sections:

1. **Latency Timeline** slide bar with **Move** and **Zoom** buttons—this timeline shows the entire recording in its full length (the Latency Timeline works the same on all tabs that start with the name *Latency*, see Using the Latency Timeline Slide Bar for more information).
2. **Filter column**—see Filtering Information on how to use this function.
3. **Thread list**—a graphic representation of all threads in the recorded JRA file.

4. **Thread magnifyer** slide bar—this slide bar lets you magnify the thread you are studying. This way you will better see each event within the thread.
5. **Show GC backdrop** and **Show only Operative Set** options—the **Show GC backdrop** option allows you to see each garbage collection as fine lines behind each thread. The **Show only Operative Set** option allows you to concentrate on studying the events that you have chosen for your operative set (see What is an Operative Set? for a description of the operative set and how to use it).

Using the Latency Timeline Slide Bar

Depending on how long your JRA recording is, the Threads graph can be quite cluttered to view in its full lengths due to all events. Therefore, you can refocus and minimize the amount of data displayed in the charts by using the Latency Timeline. You can also

Use the move and zoom buttons to refocus in the Latency graph

Move the slide bar from the top of the tab to the bottom.

The different ways to use the Latency Timeline are described in the following topics:

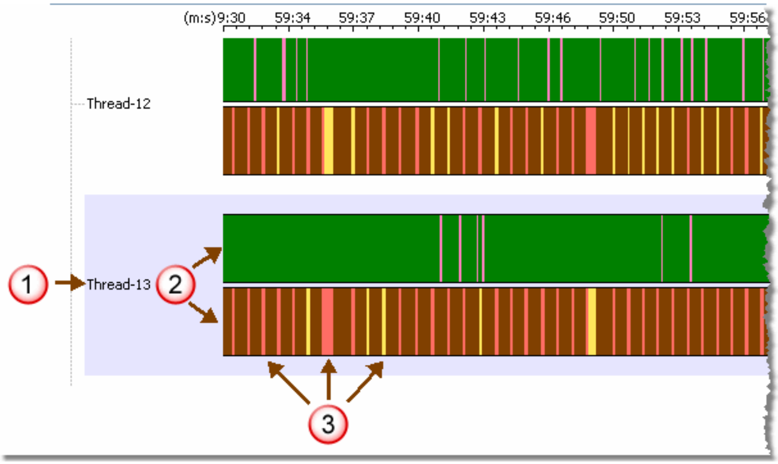
- To decrease the time span on a latency tab
- To refocus using the timeline slide bar
- To move and zoom using the move and zoom buttons
- To reposition the timeline slide bar

Understanding the Different Parts of a Thread Image

A thread contains information on the levels and events that have been taking place during the recording. Figure 18-2 illustrates how a thread looks when it is zoomed in and magnified.

For information on how to zoom in and magnify a thread, see Magnifying a Thread.

Figure 18-2 Magnifying a and zooming a thread



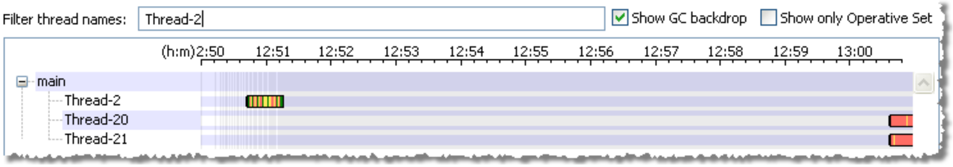
The following information becomes visible when magnifying a thread:

1. The thread itself. This is triggered by the *Producer*, i.e. the part of the system that produced an event for that thread, for example, JRockit.
2. The different levels of the thread. These are imaginary levels and depict that an event can only take place in one level at a time within the same thread.
3. The events that have taken place in the thread. Each event type has its own color (can be customized). When you hoover over an event, you will get more information about that event.

Filtering on Thread Names

The **Filter thread names** field lets you filter our the threads that you are interested in viewing. The example in Figure 18-3, depicts how it looks when you have typed in Thread-2. The Threads graph show the threads starting with the number 2 only, which can make viewing easier.

Figure 18-3 Filtering threads



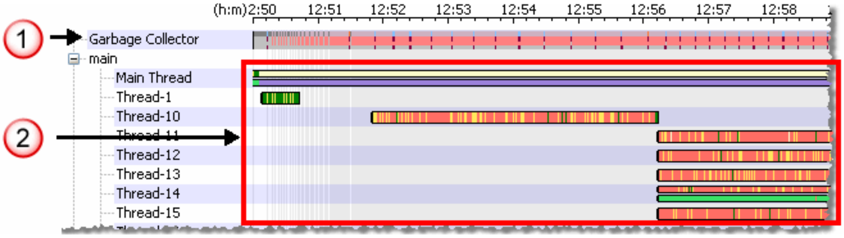
Filter thread names: Thread-2

☑ Show GC backdrop ☐ Show only Operative Set

What Does the Threads Chart Contain?

The **Threads** chart lists all threads that have been active during the recording. The threads are quite colorful at a first glance where every color represents an event. Figure 18-4 shows an example of threads. The garbage collections are located at the top of the list (marked 1 in Figure 18-4) and each thread is located below its thread group in alphabetical order (marked 2 in Figure 18-4).

Figure 18-4 Threads list



Each thread in the Threads list contains events. A thread can also contain different levels within the same thread (see Thread-14 in Figure 18-4 for an example of levels). To see the actual events with some granularity, you can magnify the thread itself (see To zoom in on a thread by using the magnifyer slide bar) and decrease the time span of the thread you are monitoring (see To decrease the time span on a latency tab).

You can also view specific properties for each event as described in About the Properties Tab or Hovering Over an Event.

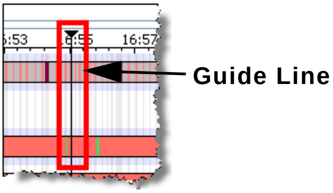
Correlating Events on Threads

You can easily correlate latency events that occur on non-adjacent threads by using a guide line that lays over the graph. For example, if you have 10 threads in your application and you want to correlate the time when a latency event happens in thread 2 (which will appear toward the top of the screen) with another latency event that happens in thread 9 (toward the the bottom of the screen), you would do the following:

1. On the elapsed time bar at the top of the Thread List, click the point in time for which you want to correlate latency events.

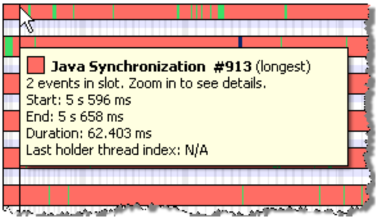
   The guide line will appear (Figure 18-5).

Figure 18-5 Latency events correlation guideline



2. Scroll down to the first event you want to correlate and hover over the intersection of the thread event and the guide line to display event information (Figure 18-6).

Figure 18-6 Thread information displayed



🟥 **Java Synchronization #913** (longest)
2 events in slot. Zoom in to see details.
Start: 5 s 596 ms
End: 5 s 658 ms
Duration: 62.403 ms
Last holder thread index: N/A

3. Scroll down to the next event you want to correlate and hover over the intersection of the thread event and the guide line to display event information

4. To clear the guide line from the graph, simply click the top of it (the black triangle).

Magnifying a Thread

To get a better view at the events within a thread, you will probably need to magnify the thread you are monitoring. There are two ways to better see events within a thread: magnify the thread or zoom in on the time span that is used.

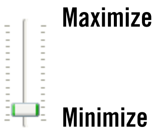
This section explains how to magnify a thread:

- To zoom in on a thread by using the magnifyer slide bar

To zoom in on a thread by using the magnifyer slide bar

1. Click and hold the **Thread magnifyer** slide bar (Figure 18-7).

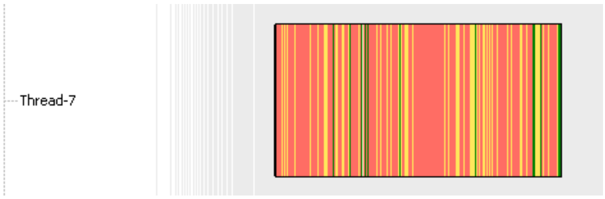
Figure 18-7 Thread magnifyer slide bar



2. Slide up to magnify and down to minimize the thread size.

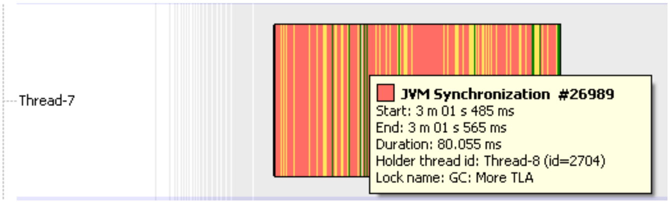
   Figure 18-8 shows a thread that has been magnified to its maximum size.

Figure 18-8 Magnified thread



3. Slide up or down, using the side scroll bars, to find the thread you want to study.
4. Hoover with the mouse over the thread, you will see details for each event in that thread (Figure 18-9).

Figure 18-9 Magnified event



🟥 JVM Synchronization #26989
Start: 3 m 01 s 485 ms
End: 3 m 01 s 565 ms
Duration: 80.055 ms
Holder thread id: Thread-8 (id=2704)
Lock name: GC: More TLA

As you can see, the events appear as large chunks were there are many of the same type and each event can be hard to see.

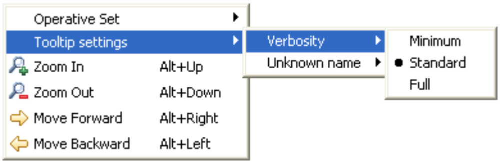
Hovering Over an Event

The default tooltip setting for hovering over an event is to display the standard information (minimized information plus holder thread and lock name). You can change the amount of information displayed in the tooltip.

To change the tooltip setting

1. Right-click anywhere in the Threads chart.
2. Click Tooltip settings > Verbosity.

Figure 18-10 Tooltip setting



3. Select a tooltip granularity.
   - **Minimum**: shows start time, end time, and duration.
   - **Standard** (default): shows start time, end time, duration, holder thread, and lock name.
   - **Full**: shows start time, end time, duration, holder thread, lock name, and stack trace.

Showing Garbage Collection Backdrop

The **Show GC backdrop** function is a helpful feature that lets you see when and where a garbage collection occurs. You will get the best visual effect of the garbage collections if you zoom in on the threads you are monitoring. The garbage collection backdrop lines might otherwise become more of a light raster in the background than helpful lines.

To turn on/off the GC backdrop lines

- Click the **Show GC backdrop** option (marked 5 in Figure 18-1) to turn on/off the GC backdrop lines.

Latency Traces Information

The **Latency Traces** tab contains a list of all methods that contain events with latencies. The method traces with the most latencies are listed first. The **Latency Traces** table can be customized to display specific packages, classes, and methods.

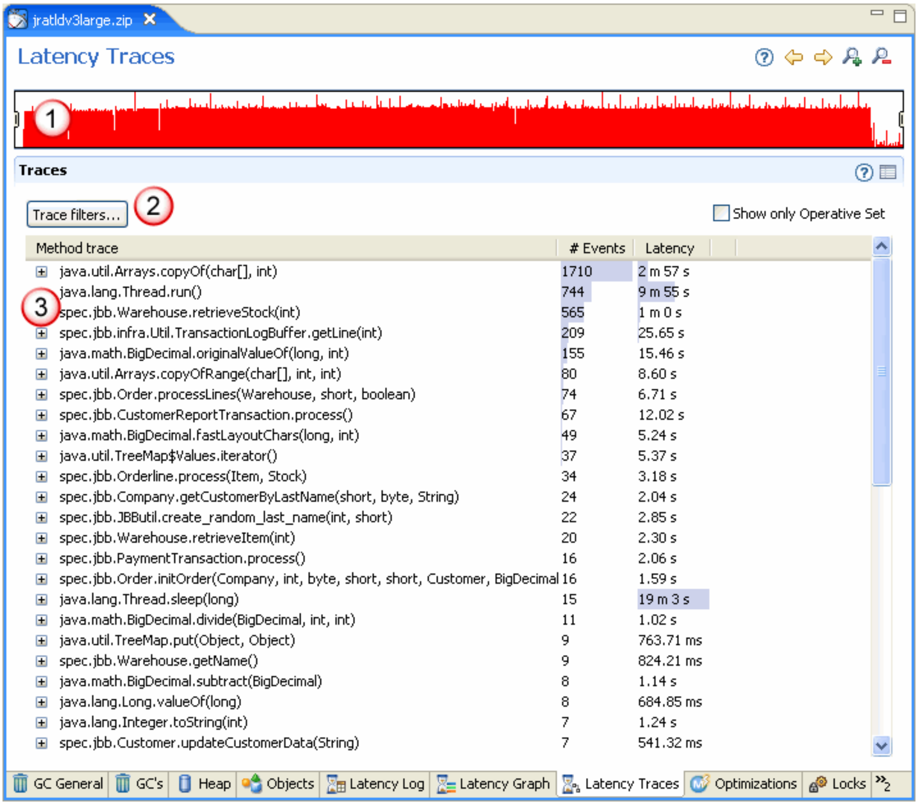
This section is divided into the following topics:

- Getting Familiar with the Latency Traces Tab
- Setting Trace Filter

Getting Familiar with the Latency Traces Tab

The **Latency Traces** tab (Figure 19-1) lists methods with most amount of events and the longest latencies.

Figure 19-1 Latency Traces tab



☐ Show only Operative Set

Method trace	# Events	Latency
java.util.Arrays.copyOf(char[], int)	1710	2 m 57 s
java.lang.Thread.run()	744	9 m 55 s
spec.jbb.Warehouse.retrieveStock(int)	565	1 m 0 s
spec.jbb.infra.Util.TransactionLogBuffer.getLine(int)	209	25.65 s
java.math.BigDecimal.originalValueOf(long, int)	155	15.46 s
java.util.Arrays.copyOfRange(char[], int, int)	80	8.60 s
spec.jbb.Order.processLines(Warehouse, short, boolean)	74	6.71 s
spec.jbb.CustomerReportTransaction.process()	67	12.02 s
java.math.BigDecimal.fastLayoutChars(long, int)	49	5.24 s
java.util.TreeMap$Values.iterator()	37	5.37 s
spec.jbb.Orderline.process(Item, Stock)	34	3.18 s
spec.jbb.Company.getCustomerByLastName(short, byte, String)	24	2.04 s
spec.jbb.JBButil.create_random_last_name(int, short)	22	2.85 s
spec.jbb.Warehouse.retrieveItem(int)	20	2.30 s
spec.jbb.PaymentTransaction.process()	16	2.06 s
spec.jbb.Order.initOrder(Company, int, byte, short, short, Customer, BigDecimal	16	1.59 s
java.lang.Thread.sleep(long)	15	19 m 3 s
java.math.BigDecimal.divide(BigDecimal, int, int)	11	1.02 s
java.util.TreeMap.put(Object, Object)	9	763.71 ms
spec.jbb.Warehouse.getName()	9	824.21 ms
java.math.BigDecimal.subtract(BigDecimal)	8	1.14 s
java.lang.Long.valueOf(long)	8	684.85 ms
java.lang.Integer.toString(int)	7	1.24 s
spec.jbb.Customer.updateCustomerData(String)	7	541.32 ms

The **Latency Traces** tab is divided into the following sections:

1. **Latency Timeline** slide bar—this timeline shows the entire recording in its full length (the Latency Timeline works the same on all tabs that start with the name *Latency*, see Using the Latency Timeline Slide Bar for more information).
2. **Trace filters** button—this button allows you to add and remove packages, classes, and methods in the Traces table.
3. **Traces table**—the Traces table lists the packages and their events. The color coding of the **# Events** and **Latency** columns gives you an overview of which package contains the events that have the greatest latencies.

Setting Trace Filter

Using filters is a great way to minimize the amount of data that is shown in the Latency Traces table. The available trace filter is quite powerful with capabilities to filter on packages, classes, and methods. You can create your own filter profile.

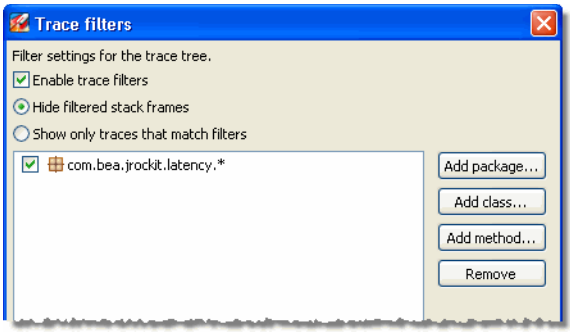
The **Latency Traces** tab has a powerful filtering function that allows you to easily filter out packages, classes, and methods from the **Latency Traces** table. That way you will get a better overview of the exact methods you want to study. You can also decide if you want to show or hide the stack frames that matches the filter.

To add a package, class, or method

1. Click the **Trace filters** button.

   The **Trace filters** window opens (Figure 19-2).

Figure 19-2 Trace filters window

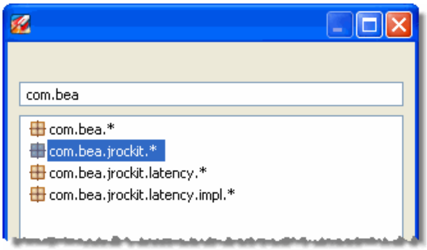


Filter settings for the trace tree.
☑ Enable trace filters
● Hide filtered stack frames
○ Show only traces that match filters
☑ com.bea.jrockit.latency.*

2. Click either the **Add package**, **Add class**, or **Add method** button.

   The **Add** window opens (Figure 19-3 shows how to add a package).

Figure 19-3 Add package window



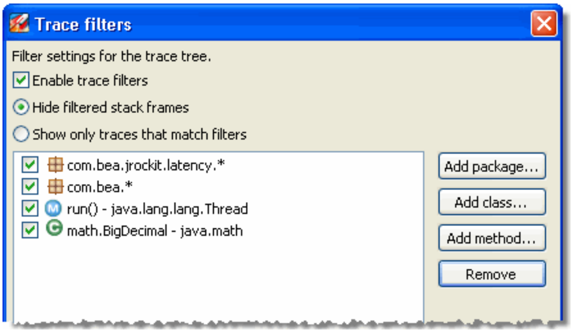
com.bea

com.bea.*
com.bea.jrockit.*
com.bea.jrockit.latency.*
com.bea.jrockit.latency.impl.*

3. Type in the prefix of the package, class, or method name, for example *com.bea* if you are adding a package, to quickly find what you are looking for.
4. Select the package, class, or method you want to use as a filter.

   The selected package now appears in the **Trace filters** window (Figure 19-4).

Figure 19-4 Trace filter with package, class, and method



Filter settings for the trace tree.
☑ Enable trace filters
● Hide filtered stack frames
○ Show only traces that match filters

☑ com.bea.jrockit.latency.*
☑ com.bea.*
☑ run() - java.lang.lang.Thread
☑ math.BigDecimal - java.math

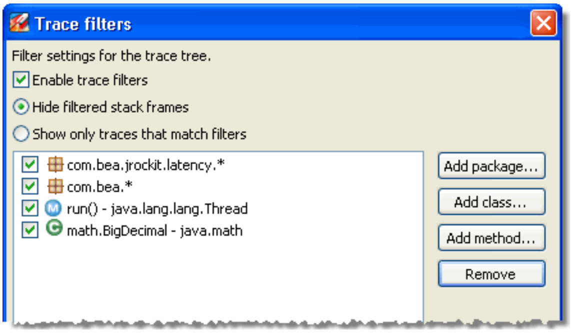
5. Select one of the following options:
   - **Hide filtered stack frames**—you will not see the stack frames that matches the selected filter.
   - **Show only traces that match filters**—you will see only the traces containing stack frames that matches the selected filter.
   - **Enable trace filters**—turns the filter function on when selected.
6. Click **OK**.

To remove a package, class, or method

1. Click the **Trace filters** button.

The **Trace filters** window opens (Figure 19-5).

Figure 19-5 Trace filters window



Filter settings for the trace tree.

☑ Enable trace filters

● Hide filtered stack frames

○ Show only traces that match filters

☑ com.bea.jrockit.latency.*

☑ com.bea.*

☑ run() - java.lang.lang.Thread

☑ math.BigDecimal - java.math

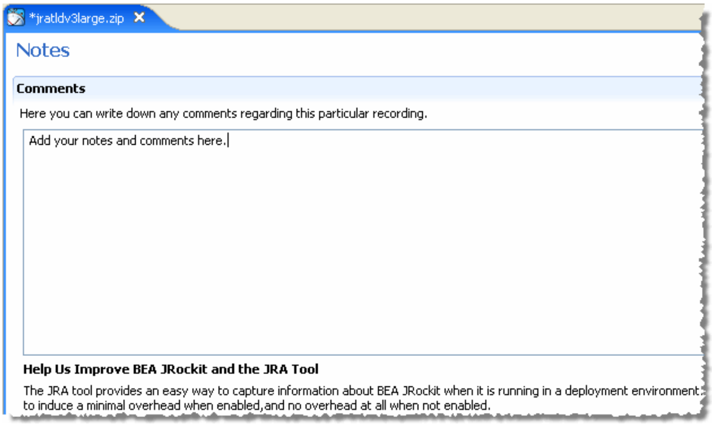
2. Select the package, class, or method you want to remove.
3. Click **Remove**.
4. Click **OK**.

Note: You can also deselect the **Enable trace filters** function to disable the filter.

Adding Comments and Notes to a Recording

The JRA Tool is equipped with a small text editor where you can add notes and 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 (Figure 20-1).

Figure 20-1 The Notes Tab



Notes

Comments

Here you can write down any comments regarding this particular recording.

Add your notes and comments here.

Help Us Improve BEA JRockit and the JRA Tool

The JRA tool provides an easy way to capture information about BEA JRockit when it is running in a deployment environment.
to induce a minimal overhead when enabled,and no overhead at all when not enabled.

To add a note

1. Enter a description of you application in the text field.
2. To save the message as part of the recording, use one of the options described in Table 20-1.

Table 20-1 Saving a recording

To save the recording...	Do this...
Under its orginal name	Select **File>Save**. The comments will be saved in a file.
Under a new name	1. Select **File>Save as...** The Save as dialog box appears 2. Open the folder into whch you want to save the recording and enter the name you under which you want to save the recording. 3. Click **Save**.

4. Close the JRA recording.