

Oracle® JRockit Mission Control

Introduction to Oracle JRockit Mission Control

3.1.0

April 2009

Copyright © 2007, 2008, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Introduction to Oracle JRockit Mission Control Client

Supportability Statement	1-2
Installation Information	1-2
Starting the JRockit Mission Control Client	1-2
JRockit Mission Control FAQ	1-2
JRockit Mission Control Documentation	1-4
JRockit Mission Control Support	1-5
Is There a Forum Where I Can Discuss JRockit Mission Control?.	1-5
Giving Feedback the JRockit Mission Control Development Team	1-5

Accessibility Notes for JRockit Mission Control Client

Screen Readers	2-1
JRockit Mission Control Accessibility Mode	2-2
Workarounds	2-3
Abbreviations Used in the JRockit Mission Control Client	2-4

Abbreviations and Acronyms

JRockit Mission Control Communications

JRockit Mission Control Client Communications Overview	4-1
J2SE 1.4	4-2
J2SE 5.0 and Later	4-3
All Versions	4-3

Integration with the Eclipse IDE

- Benefits of the Integration 5-1
- Differences between the Eclipse Version and the RCP Version 5-2
- Making the JRockit JVM Your JVM. 5-2
- Selecting a Perspective 5-4
- Jumping to Application Source 5-7

Overview of JRockit Mission Control Client 3.1.0

- New Features in JRockit Mission Control 3.1.0 6-1
- Architectural Overview of JRockit Mission Control 3.1.0 6-2
- Starting JRockit Mission Control 6-3
- The JRockit Browser 6-3
- The JRockit Management Console 6-4
- The JRockit Runtime Analyzer (JRA). 6-5
- The JRockit Memory Leak Detector 6-6

Overview of JRockit Mission Control 2.0

- Architectural Overview of JRockit Mission Control 2.0. 7-1
- Starting JRockit Mission Control 7-3
- The JRockit Browser 7-3
- The JRockit Management Console 7-3
- The JRockit Runtime Analyzer (JRA). 7-4
- The JRockit Memory Leak Detector 7-5

Overview of Oracle JRockit 1.0

- Architectural Overview of JRockit Mission Control 1.0. 8-1
- The JRockit Management Console 8-2
- The JRockit Runtime Analyzer (JRA). 8-3
- The JRockit Memory Leak Detector 8-3

Oracle JRockit Mission Control Client Use Cases

- Analyzing System Behavior with the JRockit Management Console 9-1
- Analyzing System Problems with the JRockit Runtime Analyzer 9-12
- Detecting a Memory Leak 9-24

Introduction to Oracle JRockit Mission Control Client

Oracle JRockit Mission Control Client is a tools suite introduced with Oracle JRockit JVM R26.0.0. It includes tools to monitor, manage, profile, and eliminate memory leaks in your Java application without introducing the performance overhead normally associated with these types of tools.

The JRockit Mission Control's low performance overhead is a result of using data collected as part of the JRockit JVM's normal adaptive dynamic optimization. This also eliminates the problem with the Heisenberg anomaly that can occur when tools using byte code instrumentation alters the execution characteristics of the system. The JRockit Mission Control functionality can always be available on-demand and the small performance overhead is only in effect while the tools are running.

This section contains information on the following subjects:

- [Supportability Statement](#)
- [Installation Information](#)
- [Starting the JRockit Mission Control Client](#)
- [JRockit Mission Control FAQ](#)
- [JRockit Mission Control Documentation](#)
- [JRockit Mission Control Support](#)
- [Is There a Forum Where I Can Discuss JRockit Mission Control?](#)
- [Giving Feedback the JRockit Mission Control Development Team](#)

Supportability Statement

JRockit Mission Control Client connects to all versions of the JRockit JVM from R26.4 up through the release with which it shipped. Connections to older versions of JRockit JVM might be possible, but are not officially supported.

JRockit Mission Control Client will open JRA-recordings created by all JRockit JVM versions from R26.4 up through the release with which it shipped. Some features (for example, latency recordings) might not be available since they were not implemented in the older recordings.

Installation Information

Installation instructions can be found in [Installing Oracle JRockit Mission Control](#).

Starting the JRockit Mission Control Client

The JRockit Mission Control Client executable is located in `JROCKIT_HOME/bin`. If this directory is on your system path, you can start JRockit Mission Control by simply typing `jrmc` in a command (shell) prompt.

Otherwise, you have to type the full path to the executable file, as shown below:

```
JROCKIT_HOME\bin\jrmc.exe (Windows)
```

```
JROCKIT_HOME/bin/jrmc (Linux)
```

On Windows installations, you can start JRockit Mission Control from the **Start** menu.

JRockit Mission Control FAQ

This topic lists and provides answers to questions frequently asked about JRockit Mission Control.

- [I can not connect the JRockit Mission Control Client. What could be the problem?](#)
- [When attempting to connect to JRockit Mission Control I get a stack trace indicating that JRockit Mission Control attempts to communicate with a strange IP or host name.](#)
- [I'm getting exceptions during startup about classes not being found](#)
- [JRockit Mission Control can't find any local JVMs](#)
- [Why can't I see any Method Profiling information in my JRA recording?](#)

- [When using the Memory Leak Detector, nothing happens in the growth column of the trend table](#)

I can not connect the JRockit Mission Control Client. What could be the problem?

Consider the following:

- Are you using the correct protocol?

The easiest way is to ensure that you are using the same version of the JRockit JVM you want to monitor as the JVM running the JRockit Mission Control client. If that is not an option, you can use the radio buttons in the connection dialog box in the JRockit Mission Control Client to select which protocol to use: 1.4 will select RMP and 1.5 and later will select JMXRMI.

For earlier versions of JRockit Mission Control these radio buttons don't exist and, to make a JRockit JVM 5.0 instance connect to a 1.4 version, you must explicitly specify the JMX Service URL. The format of the service URL is:

```
service:jmx:rmp://<hostname>:<port>
```

for example:

```
service:jmx:rmp://localhost:7091
```

- Are the correct ports opened?

JMX over RMI uses two ports and that one of the ports will not be known beforehand.

- Is the communication caught in the firewall?

Please see JRockit Mission Control Communications for more information.

When attempting to connect to JRockit Mission Control I get a stack trace indicating that JRockit Mission Control attempts to communicate with a strange IP or host name.

Sometimes RMI can have a problem determining which address to use. This can happen because of

- Access restrictions in the Security manager.
- The machine being multihomed and RMI picking the wrong interface.
- A misconfigured hosts file or a number of different network related configuration problems.

If all else fails you can try specifying the `java.rmi.server.hostname` system property. Please note that this can affect applications running in the JVM.

I'm getting exceptions during startup about classes not being found

Make sure you are using the proper launcher to start up the JRockit Mission Control Client. You must only use `JROCKIT_HOME/bin/jrmc`.

JRockit Mission Control can't find any local JVMs

Make sure you are using the proper launcher to start up the JRockit Mission Control Client. You must only use `JROCKIT_HOME/bin/jrmc`.

Why can't I see any Method Profiling information in my JRA recording?

In versions prior to 3.1.0, by default the JRockit Mission Control Client didn't show tabs if data had not been recorded for them. If you are using an earlier version of the product, ensure that method profiling was enabled for your JRA recording and that the application was under load. If the JRockit JVM is spending most of the time with none of the threads doing any work, no samples will be recorded. If you still want to create a JRA recording with method sampling and a low load, try increasing the sampling frequency.

When using the Memory Leak Detector, nothing happens in the growth column of the trend table

The algorithm needs at least three data points to kick in and the data is collected as part of the old space mark phase of the garbage collection. If you see no data, possibly not enough garbage has been collected for these collections to occur. To speed up the process, try clicking the garbage can in the toolbar of the Memory Leak application to force three successive garbage collections, with a brief pause in between each collection.

JRockit Mission Control Documentation

Documentation for JRockit Mission Control 1 is available on eDocs:

<http://edocs.bea.com/jrockit/tools/index.html>

Documentation for JRockit Mission Control 2 and 3 is available both as online help with the installation of the tool and as PDFs on eDocs:

<http://edocs.bea.com/jrockit/tools/index.html>

Note: The PDFs for JRockit Mission Control 2 on eDocs are not the full versions. Look in the Help menu for full documentation.

JRokit Mission Control Support

You are entitled to support if you have an Enterprise licence.

Is There a Forum Where I Can Discuss JRokit Mission Control?

If you have any questions you are welcome to share them in the Oracle JRokit forum, which is monitored by the Oracle JRokit engineering team. To access the news group, go to:

<http://forums.oracle.com/forums/forum.jspa?forumID=561>

Giving Feedback the JRokit Mission Control Development Team

If you have any suggestions about how to improve the JRokit Mission Control or information on how it is most commonly used in your development environments, we would be grateful to receive your input. This information helps us improve these tools going forward.

Please send an e-mail with feedback and your ideas on how we can improve Oracle JRokit Mission Control to:

`jrokit-improve@oracle.com`

We will consider this feedback, along with other ideas we collect, to improve the product and make it even easier to use. Oracle's goal is to simplify the tasks and provide excellent functionality.

Introduction to Oracle JRockit Mission Control Client

Accessibility Notes for JRockit Mission Control Client

Oracle is dedicated to providing high quality information technology that is accessible to people with disabilities. To this end, Oracle has undertaken a substantial project to ensure the accessibility of Oracle JRockit Mission Control Client. Oracle is implementing these enhancements and will continue to address all accessibility issues that come to its attention.

This document includes information on the following subjects:

- [Screen Readers](#)
- [JRockit Mission Control Accessibility Mode](#)
- [Workarounds](#)
- [Abbreviations Used in the JRockit Mission Control Client](#)

Screen Readers

Oracle supports a number of different screen readers, technology that translates screen-based information into spoken word to assist vision-impaired users.

Configuration options are currently available for the JAWS™ screen reader produced by Freedom Scientific, Inc. For information on configuring this product, please refer to the [Freedom Scientific screen reader website](#), at:

<http://www.freedomscientific.com/documentation/screen-readers.asp>

Note: If you are using JAWS, be aware that tab/page switching does not work as expected. Please refer to [Switching Between Tabs/Pages](#) for a workaround.

JRockit Mission Control Accessibility Mode

Throughout JRockit Mission Control Client, dials and charts are used to display performance data. For most users, these charts provide a valuable graphical view of the data that can reveal trends and help identify minimum and maximum values for performance metrics; however, charts do not convey information in a manner that can be read by a screen reader. To remedy this problem, you can configure JRockit Mission Control Client *accessibility mode* to provide dial and chart data in tabular format.

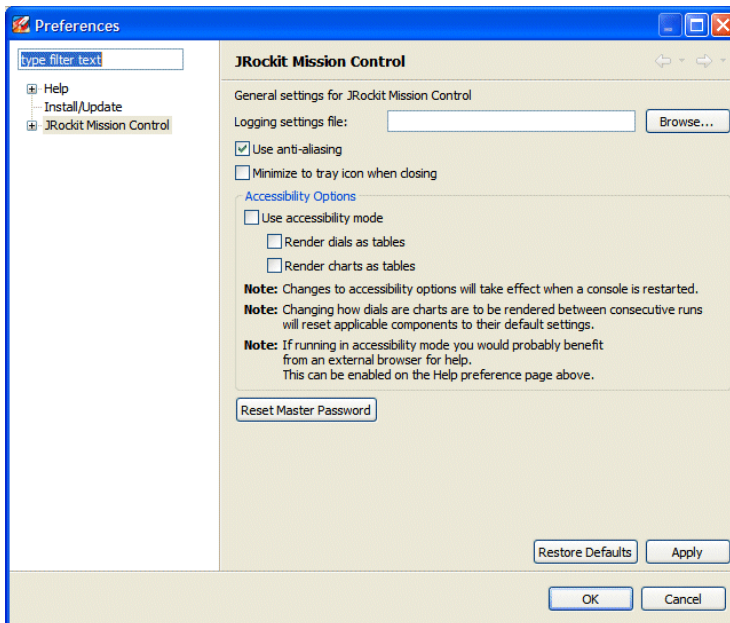
Configuring the Accessibility Mode

The accessibility mode is configured from JRockit Mission Control Client preferences window. To enable it, do the following:

1. With JRockit Mission Control Console running, select **Windows > Preferences...**

The Preferences window appears (Figure 1-1).

Figure 1-1 Preferences Window



2. In the **Accessibility Options** pane, select **Use accessibility mode** and, under that, select the accessibility options you want to enable:
 - **Render dials as tables** presents data shown on the velocimeters as table data.
 - **Render graphs as tables** presents data shown in graphs as table data.
3. Select **OK**.
4. Stop and restart the Console.

Note: If you configuring the accessibility mode before actually starting the console, the new configuration will appear when you start the console.

For Additional Information

For important information on using a screen reader to read table data, please refer to [Reading Table Data with a Screen Reader](#).

Workarounds

This section contains additional instructions for enhancing your experience with JRockit Mission Control's accessibility features. These instructions include:

- [Switching Between Tabs/Pages](#)
- [Reading Table Data with a Screen Reader](#)
- [Reading Console General/Overview Tab Data](#)
- [Resizing Online Help Text](#)

Switching Between Tabs/Pages

When reaching a tab component in the JRockit Mission Control GUI, JAWS erroneously tells the user “to switch pages, press Ctrl+Tab”. The correct way to switch between tabs/pages is to use the left/right arrow keys.

Reading Table Data with a Screen Reader

To read JRockit Mission Control table data more efficiently with screen reading software, copy and paste the table data into a text editor and read it from there. To copy and paste, do the following:

1. Right-click the table you want to read to open the context menu.
2. Select all items by selecting **Select All**.
3. Select **Copy**.
4. Paste the text in a text editor.

Reading Console General/Overview Tab Data

To read data from the JRockit Mission Control Console, General/Overview tab with screen reading software, select **Window > Preferences > JRockit Mission Control** and choose **Render dials as tables** and **Render charts as tables** in the right panel.

Resizing Online Help Text

Vision-impaired users might find it difficult to read the online help documents in the standalone (RCP) version of JRockit Mission Control unless the text size is increased. If they need to change the font size, they must view the help in another browser.

JRockit Mission Control will use the default web browser specified in the operating system; you can't specify a different one from within JRockit Mission Control. How you specify a browser in the operating system depends on the version of the operating system.

To change the default viewer for JRockit Mission Control Help, do the following:

1. Select **Window > Preferences**

The Preferences dialog box appears.

2. Under **Specify how help information is displayed**, select **Use external browser**.
3. Click **Apply** or **OK**.

Note: Due to a limitation in the Eclipse help viewer, currently you cannot resize its text. If you need to resize the text the workaround is to use an external browser.

Abbreviations Used in the JRockit Mission Control Client

JRockit Mission Control Client uses a number of abbreviations and acronyms to save space on the GUI and in the documentation. Wherever possible, these abbreviations and acronyms are spelled out upon first usage; however, that is not always possible. Therefore, we have included a list of all common JRockit Mission Control Client abbreviations and acronyms in [Abbreviations and Acronyms](#).

Abbreviations and Acronyms

This document lists the abbreviations and acronyms used in Oracle JRockit Mission Control Client ([Table 2-1](#)).

Table 2-1 Abbreviations and Acronyms Used in Oracle JRockit Mission Control Client

Abbreviation	Meaning
GC	Garbage Collection (memory management)
JDK	Java Development Kit
JDP	JRockit Discovery Protocol
JIT	Just In Time (compilation)
JMX	Java Management Extensions
JRA	JRockit Runtime Analyser
JVM	Java Virtual Machine
MBean	Managed Bean (Java)
MLP	Memory Leak Protocol
RCP	Rich Client Platform (Eclipse)
RMI	Remote Method Invocation (Java)

Abbreviations and Acronyms

Table 2-1 Abbreviations and Acronyms Used in Oracle JRockit Mission Control Client

Abbreviation	Meaning
RMP	Rockit Management Protocol (communication)
SSL	Secure Sockets Layer (communication)
TLA	Thread Local Area (memory management)

JRokit Mission Control Communications

Depending upon which J2SE version on which you are running the Oracle JRokit JVM, certain aspects of the communications protocols will differ. This chapter describes the protocols and their differences resultant from the different J2SE version.

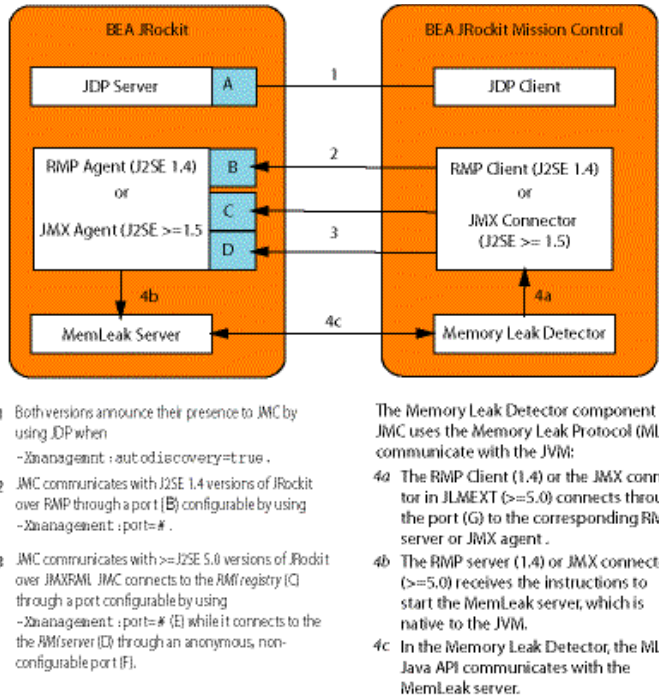
This section includes information on the following subjects:

- [JRokit Mission Control Client Communications Overview](#)
- [J2SE 1.4](#)
- [J2SE 5.0 and Later](#)
- [All Versions](#)

JRokit Mission Control Client Communications Overview

[Figure 3-1](#) illustrates the communication topology for Oracle JRokit Mission Control.

Figure 3-1 JRockit Mission Control Communications Topology



J2SE 1.4

J2SE 1.4 versions of JRockit Mission Control uses RMP (Rockit Management Protocol.), an older legacy protocol that has existed since the 1.3 versions of the Oracle JRockit JDK. RMP uses a single socket. You can specify the port of the listening socket by using the `-Xmanagement:port` option; for example `-Xmanagement:port=7090`. Table 1-1 lists additional system properties you can use to further configure the agent.

Table 3-1 Additional Communication Settings for JRockit Mission Control on J2SE 1.4

System Property	Description	Default
<code>jrockit.managementserver.address</code>	Bind to a specific interface	Not enabled, listens on all interfaces)

Table 3-1 Additional Communication Settings for JRockit Mission Control on J2SE 1.4

System Property	Description	Default
<code>jrockit.managementserver.timeout</code>	Socket time-out in MS	4000
<code>jrockit.managementserver.maxconnect</code>	Maximum number of connections	4

J2SE 5.0 and Later

J2SE 5.0 and later versions of the JRockit JDK use JMXRMI (JMX over RMI). This protocol uses one port for the RMI registry, which is configured with the `-Xmanagement:port` option, and a second port (on an anonymous port) for communication with the RMI server. Note that you cannot configure the port for the RMI server; however, you can write your own agent that defines a fixed port for the RMI server. Please see the following link for further information:

<http://java.sun.com/javase/6/docs/technotes/guides/management/agent.html#dfvv>

Table 3-2 lists the options available for the `-Xmanagement` flag:

Table 3-2 -Xmanagement Option

Option	Description	Default
<code>authenticate</code>	Use password authentication	True
<code>ssl</code>	Use secure sockets layer	True
<code>port</code>	What port to use for the RMI registry	7091

For a more comprehensive discussion on what these options mean, please see:

<http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html>.

All Versions

For all J2SE versions, you can use the `-Xmanagement` option autodiscovery to make the JRockit JVM use the JRockit Discovery Protocol (JDP) to announce its presence; for example `-Xmanagement:autodiscovery=true`.

Table 3-3 lists additional system properties you can use to control the behavior of the JDP server:

Table 3-3 System Properties Used to Control the JDP Server

System property	Description	Default
<code>jrokit.managementserver.discovery.period</code>	The time to wait between multicasting the presence in ms	5000
<code>jrokit.managementserver.discovery.ttl</code>	The number of router hops the packets being multicasted should survive	1
<code>jrokit.managementserver.discovery.address</code>	The multicast group/address to use	232.192.1.212
<code>jrokit.managementserver.discovery.targetport</code>	The target port to broadcast	7090(1.4)/7091(1.5)

All versions of JRokit Mission Control also employ an additional protocol when using the Memory Leak Detector. The memleak server is not written in Java; rather it is an integral part of the JVM. This is because a potential use case for the memleak server is to optionally be able to start it when an out of memory condition occurs in the JVM. When such a condition occurs, it is impossible to execute Java code because no heap would be available.

MLP (MemLeak Protocol) is used by the native memleak server during a memleak session. JRokit Mission Control communicates over RMP (1.4) or JMXRMI (5.0 and higher) to ask the Oracle JRokit JVM to start up the server. You can configure the port on which you want to start the memleak server on, and to use for the session, by using Oracle JRokit Mission Control preferences. See [Figure 3-1](#) for schematic of this process.

Integration with the Eclipse IDE

In addition to the standalone Rich Client Platform (RCP) version of Oracle JRockit Mission Control 3.1.0, the toolset is also available as a plug-in to the Eclipse IDE (Eclipse 3.3 or above). This version of JRockit Mission Control provides seamless integration of JRockit Mission Control's application profiling and monitoring toolset with the Eclipse development platform. By integrating the JRockit Mission Control Client with Eclipse, you can combine the features of Eclipse with the power toolset in JRockit Mission Control.

This document describes this integration and provides instructions for using the special functionality enabled by integrating the JRockit Mission Control Client with Eclipse. The topics include:

- [Benefits of the Integration](#)
- [Differences between the Eclipse Version and the RCP Version](#)
- [Making the JRockit JVM Your JVM](#)
- [Selecting a Perspective](#)
- [Jumping to Application Source](#)

Benefits of the Integration

When the JRockit Mission Control Client is run within the Eclipse IDE, you have access to IDE features that aren't otherwise available in the toolset when it is run as a standalone Rich Client Platform (RCP) application. The most significant of these features is the ability to see specific

code in the running application by opening it directly from the JRockit Mission Control Client, a function called Jump-to-Source.

The other obvious benefit of integrating the JRockit Mission Control Client with the Eclipse IDE is that it allows you to profile and monitor an application during their development phase just as you would during their production phase. This allows you to spot potential runtime problems before you actually deploy your application to production; for example, you might, while monitoring an application during its development notice a memory leak. By catching the memory leak during development, you can correct it before you migrate your application to a production environment.

Differences between the Eclipse Version and the RCP Version

Generally, the Eclipse version of the JRockit Mission Control Client works identically to the RCP version. Any component in the Eclipse version offers the same functionality and user interface as the comparable component delivered on the RCP.

The biggest difference that Eclipse version of the JRockit Mission Control Client has over the RCP version is the Jump-to-Source feature, described in [Jumping to Application Source](#). With this feature, you can not only see the name of a “problem” class or method displayed in the JRockit Mission Control Client, but you can jump from the displayed name directly to that class or method’s source, where you can evaluate the code to see what might be causing the problem. Jump-to-Source is enabled for the Management Console, the JRockit Runtime Analyzer, and the Memory Leak Detector.

Making the JRockit JVM Your JVM

While JRockit Mission Control can work with many different Java Virtual Machines, it is highly recommended that you use the Oracle JRockit JVM as your JVM when running Mission Control on the Eclipse platform. Not only will you avail yourself of the JRockit JVM’s exceptional performance, but by using this JVM, Mission Control’s autodetect feature will be enabled, which makes it simple to connect Mission Control to your locally running application.

To run Eclipse (and thus the JRockit Mission Control Client) on the JRockit JVM

1. Go to your file system browser (for example, Windows Explorer).

2. Locate your Eclipse installation folder (for example, `C:\Program Files\Eclipse`) and, with a file editor other than Notepad, open the file `eclipse.ini`. It will look something like the example in [Listing 4-1](#).

Listing 4-1 eclipse.ini Example

```
-showsplash
org.eclipse.platform
--launcher.XXMaxPermSize
256M
-vmargs
-Dosgi.requiredJavaVersion=1.5
-Xms40m
-Xmx512m
```

3. Make the following changes to `eclipse.ini`:

- Remove all flags related to non-Oracle JRockit JVMs (for example, `--launcher.XXMaxPermSize 256M`)
- On the third line down (after `org.eclipse.platform`), add the following:

```
-vm
<Full path to JRockit JVM's javaw file>
```

The full path to JRockit's `javaw` file might look like this on Windows:

```
C:\Program Files\Java\jrockit-R27.4.0-jdk1.6.0_02\bin\javaw.exe
```

or like this on Linux and Solaris:

```
$HOME/jrockit-R27.4.0-jdk1.6.0_02/bin/javaw
```

- Depending upon your particular JRockit JVM implementation and the applications running on it, you can set any valid JRockit JVM command-line option. For example, you might want to set a garbage collector that meets your system priorities by using the `-XgcPrio:` option or increase (or decrease) the initial and maximum heap size by changing the values for `-Xms` and `-Xmx`.

For more information on tuning the JVM, please refer to [Profiling and Performance Tuning](#) in the Oracle JRockit JVM *Diagnostics Guide*.

For more information on the available command-line options, please refer to the Oracle JRockit JVM [Command-Line Reference](#).

4. When you are done making the necessary changes to `eclipse.ini`, save and close the file. [Listing 4-2](#) shows an example of the `eclipse.ini` file updated to make Oracle JRockit JVM the JVM.

Listing 4-2 Updated eclipse.ini file for a Windows implementation

```
-showsplash
org.eclipse.platform
-vm
C:\Program Files\Java\jrockit-R27.4.0-jdk1.6.0_02\bin\javaw.exe
-vmargs
-Dosgi.requiredJavaVersion=1.5
-Xms256m
-Xmx512m
-XgcPrio:pausetime
```

Selecting a Perspective

A “perspective” defines a set of views and their relative positions within the Eclipse window; in other words, it is a template for graphically presenting different types of information in Eclipse. For example, the Java perspective combines views that you would commonly use while editing Java source files, while the Debug perspective contains the views that you would use while debugging Java programs.

JRockit Mission Control for Eclipse come with a predefined perspective called JRockit Mission Control. This perspective shows the JRockit Mission Control Client interface so that you can use the tools in JRockit Mission Control to profile applications as you develop them in Eclipse.

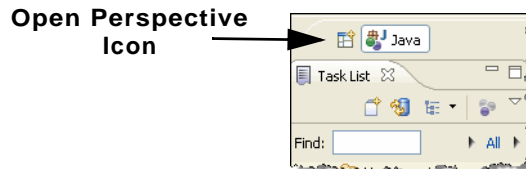
This topic will show you how:

- [To open the Mission Control Perspective](#)
- [To change perspective from Mission Control](#)
- [To reopen the Mission Control Standard Perspective](#)

To open the Mission Control Perspective

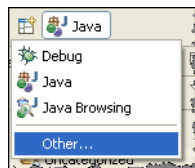
1. In top right corner of the Eclipse window, click the Open Perspective icon (Figure 4-1).

Figure 4-1 Open Perspective Icon



The Open Perspective context menu appears (Figure 4-2).

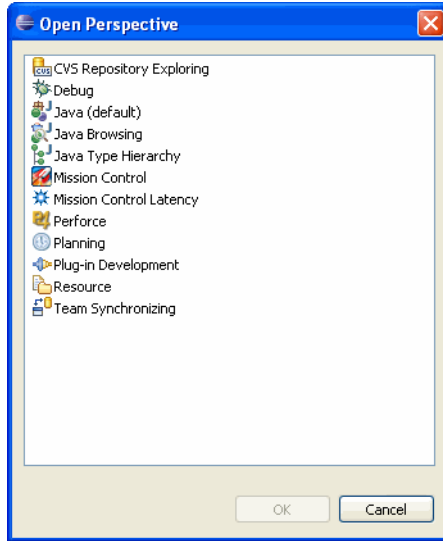
Figure 4-2 Open Perspective Context Menu



2. Select **Other...**

The Open Perspective dialog box appears (Figure 4-3).

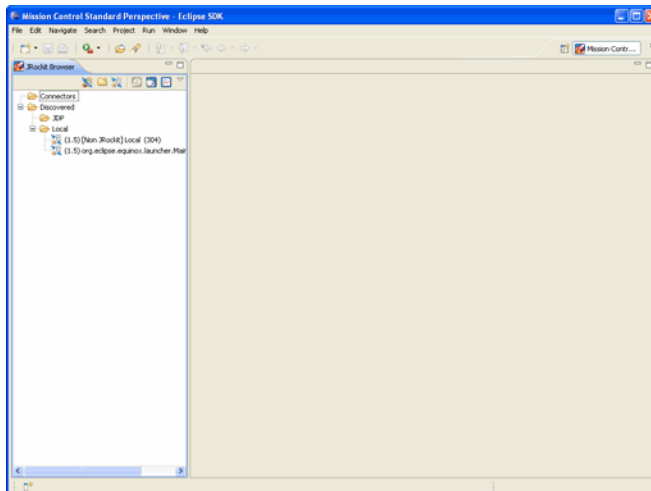
Figure 4-3 Open Perspective Dialog Box



3. Select **Mission Control** and click **OK**.

The Eclipse window reconfigures to show the Mission Control Standard Perspective (Figure 4-4).

Figure 4-4 Mission Control Standard Perspective



To change perspective from Mission Control

You can change perspectives from Mission Control to another perspective by using one of the methods described in [Table 4-1](#):

Table 4-1 Changing Perspectives

If...	Do this...
You've never opened the perspective	<ol style="list-style-type: none"> 1. Click the Open Perspective icon. 2. Either: <ul style="list-style-type: none"> – Select the perspective you want to open. – If the perspective name does not appear on the context menu, select Other... to open the Open Perspective dialog box and select the perspective from there.
You've opened the perspective before	If you've opened the perspective before, a button for that perspective will appear in the top right corner of the Standard Mission Control Perspective, near the Open Perspective icon. Simply click the button for the perspective you want to open.

To reopen the Mission Control Standard Perspective

If you have already opened the Mission Control Standard Perspective for this project, a Mission Control button will appear next to the Open Perspective button in the top right corner of the Eclipse window ([Figure 4-5](#)).

Figure 4-5 Open Standard Mission Control Perspective Button



To reopen the perspective, simply click that button.

Jumping to Application Source

When running JRockit Mission Control in an Eclipse IDE you can select a method or class and jump from the JRockit Mission Control Client directly to the source code where that method or class is declared. An editor will open up showing you the source file. Jump-to-Source is available in JRA, the Management Console and the Memory Leak Detector:

This topic contains the following information:

- [Using Jump-to-Source](#)
- [JRockit Mission Control Plug-ins with Jump-to-Source Enabled](#)

Using Jump-to-Source

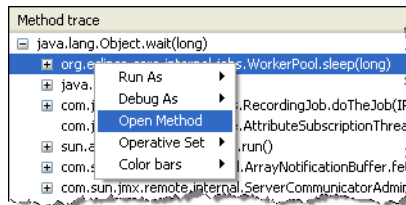
To jump from the JRockit Mission Control Client to source code

Note: The following procedure is generic. See [JRockit Mission Control Plug-ins with Jump-to-Source Enabled](#) for a list of plug-ins where this feature is enabled.

1. On the table, tree or other GUI component listing classes or methods, right-click the class or method for which you want to see the source code.

A context menu appears ([Figure 4-6](#)).

Figure 4-6 Jump to Source Command on the Context Menu



2. Select **Open Method** (or **Open Type**, if you are jumping from to a class call).

The associated source code will appear in a new editor.

JRockit Mission Control Plug-ins with Jump-to-Source Enabled

Note: This feature only works with versions of the JRockit Mission Control Client integrated into the Eclipse IDE.

[Table 4-2](#) lists the Oracle JRockit Mission Control plug-ins where Jump-to-Source is enabled.

Table 4-2 Plug-ins with Jump-to-Source Enabled

Plug-in	Component
Management Console	<ul style="list-style-type: none"> • Threads tab <ul style="list-style-type: none"> – Stack traces for selected threads • Exception Counter <ul style="list-style-type: none"> – Profiling Information table
JRA	<ul style="list-style-type: none"> • Methods Tab <ul style="list-style-type: none"> – The tables and both the trees. • GCs Tab <ul style="list-style-type: none"> – The GC method call tree for a garbage collection. • GC General Tab <ul style="list-style-type: none"> – Garbage collection call trees. • Objects Tab <ul style="list-style-type: none"> – Both Start of Recording and End of recording • Optimizations: <ul style="list-style-type: none"> – In the table. • Locks <ul style="list-style-type: none"> – Java Locks • Latency Log <ul style="list-style-type: none"> – Event Details – Event Properties – Stack Trace • Latency Log <ul style="list-style-type: none"> – Event Property Histogram, • Latency Traces <ul style="list-style-type: none"> – The trace trees.
Memory Leak Detector	<ul style="list-style-type: none"> • Trend Table • Application Stack Traces

Integration with the Eclipse IDE

Overview of JRockit Mission Control Client 3.1.0

The Oracle JRockit Mission Control Client 3.1.0 tools suite includes tools to monitor, manage, profile, and eliminate memory leaks in your Java application without introducing the performance overhead normally associated with these types of tools.

This section contains information on the following subjects:

- [Architectural Overview of JRockit Mission Control 3.1.0](#)
- [Starting JRockit Mission Control](#)
- [The JRockit Browser](#)
- [The JRockit Management Console](#)
- [The JRockit Runtime Analyzer \(JRA\)](#)
- [The JRockit Memory Leak Detector](#)

New Features in JRockit Mission Control 3.1.0

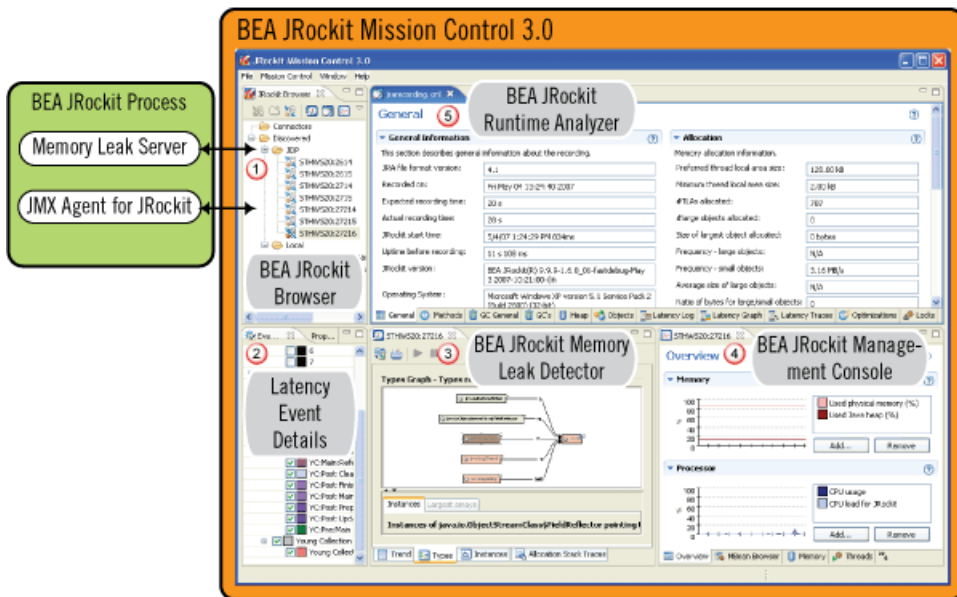
JRockit Mission Control 3.1.0 contains a large number of new features that will provide more information more seamlessly and improve the overall user experience. For descriptions of these features, please refer to [New Features and Enhancements in this Release](#) at:

http://download.oracle.com/docs/cd/E13150_01/jrockit_jvm/jrockit/tools/relnote_stools/relnotestools3.html#wp1091816

Architectural Overview of JRockit Mission Control 3.1.0

With the new Rich Client Platform (RCP) based JRockit Mission Control, you can launch the Memory Leak Detector, the JRockit Runtime Analyzer, and the JRockit Management Console from within the JRockit Mission Control. [Figure 5-1](#) depicts how JRockit Mission Control looks when all tools are loaded.

Figure 5-1 Architectural Overview of JRockit Mission Control 3.1.0



When a JRA recording is started from within the JRockit Mission Control Client, it records the status of the JRockit JVM process for the time that you have specified and creates a ZIP file containing an XML file with the recorded data and optionally a binary file with latency data together with the corresponding data producer specification files. The ZIP file is automatically opened in the JRockit Runtime Analyzer tool upon completion of the recording, valid for JDK level 1.5 and later (marked 5 in [Figure 5-1](#)). Typical information that is recorded during a JRA recording is Java heap distribution, garbage collections, method samples, and lock profiling information (optional). New for the Oracle JRockit Mission Control 3.1.0 release, is that you can also record thread latency data. When viewing Latency data in the JRA Tool, the Latency Events Details become visible (marked 2 in [Figure 5-1](#)).

To view real-time behavior of your application and of Oracle JRockit JVM, you can connect to an instance of the JRockit JVM and view real-time information through the JRockit Management Console (marked 4 in [Figure 5-1](#)). Typical data that you can view is thread usage, CPU usage, and memory usage. All graphs are configurable and you can both add your own attributes and redefine their respective labels. In the Management Console you can also create rules that trigger on certain events, for example, an mail will be sent if the CPU reaches 90% of the size.

With the JMX Agent you have access to all MBeans deployed in the platform MBean server. From these MBeans, you can read attribute information, such as garbage collection pause times.

To find memory leaks in your Java application, you connect the JRockit Memory Leak Detector to the running JRockit JVM process. The Memory Leak Detector connects to the JMX (RMP) Agent that instructs to start a Memory Leak server where all further communication takes place.

Starting JRockit Mission Control

The JRockit Mission Control Client executable is located in `JROCKIT_HOME/bin`. If this directory is on your system path, you can start the JRockit Mission Control Client by simply typing `jrmc` in a command (shell) prompt.

Otherwise, you have to type the full path to the executable file, as shown below:

```
JROCKIT_HOME\bin\jrmc.exe (Windows)
```

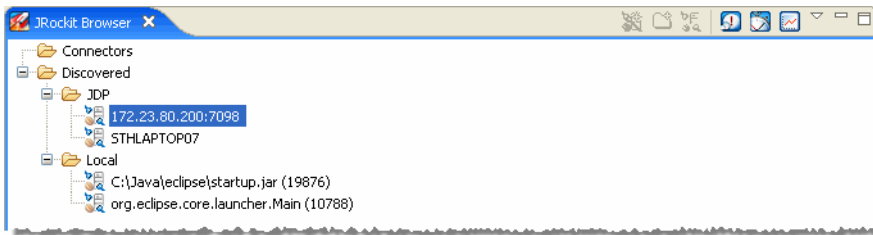
```
JROCKIT_HOME/bin/jrmc (Linux)
```

On Windows installations, you can start JRockit Mission Control from the **Start** menu.

The JRockit Browser

The JRockit Browser (see [Figure 5-2](#)) was new for the JRockit Mission Control 2.0 release. This tool allows you to set up and manage all running instances of JRockit JVM on your system. From the JRockit Browser you activate different tools, such as starting a JRA recording, connecting a Management Console, and starting memory leak detection. Each JRockit JVM instance is referred to as a *Connector*.

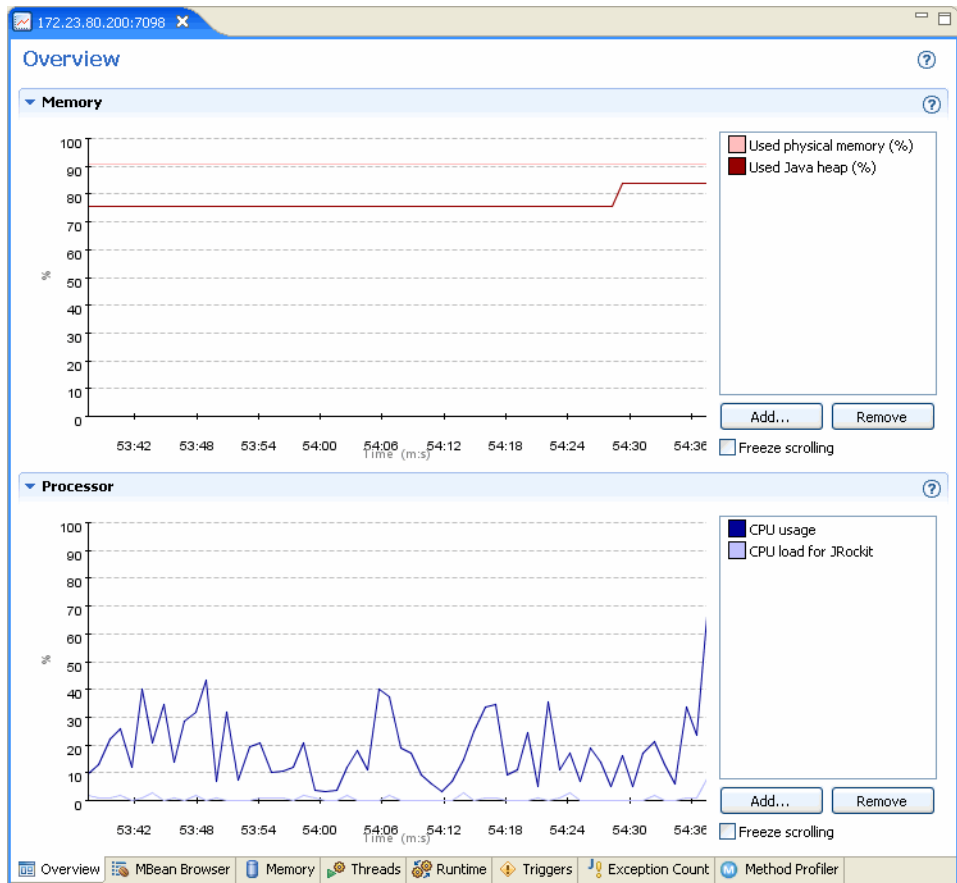
Figure 5-2 The JRockit Browser



The JRockit Management Console

The JRockit Management Console (see [Figure 5-3](#)) is used to monitor a JRockit JVM instance. Several Management Consoles can be running concurrently side by side. The tool captures and presents live data about memory, CPU usage, and other runtime metrics. For the Management Console that is connected to JRockit JDK 5.0, information from any JMX MBean deployed in the Oracle JRockit JVM internal MBean server can be displayed as well. For a Console connected to Oracle JRockit JDK 1.4, RMP capabilities are exposed by a JMX proxy. JVM management includes dynamic control over CPU affinity, garbage collection strategy, memory pool sizes, and more.

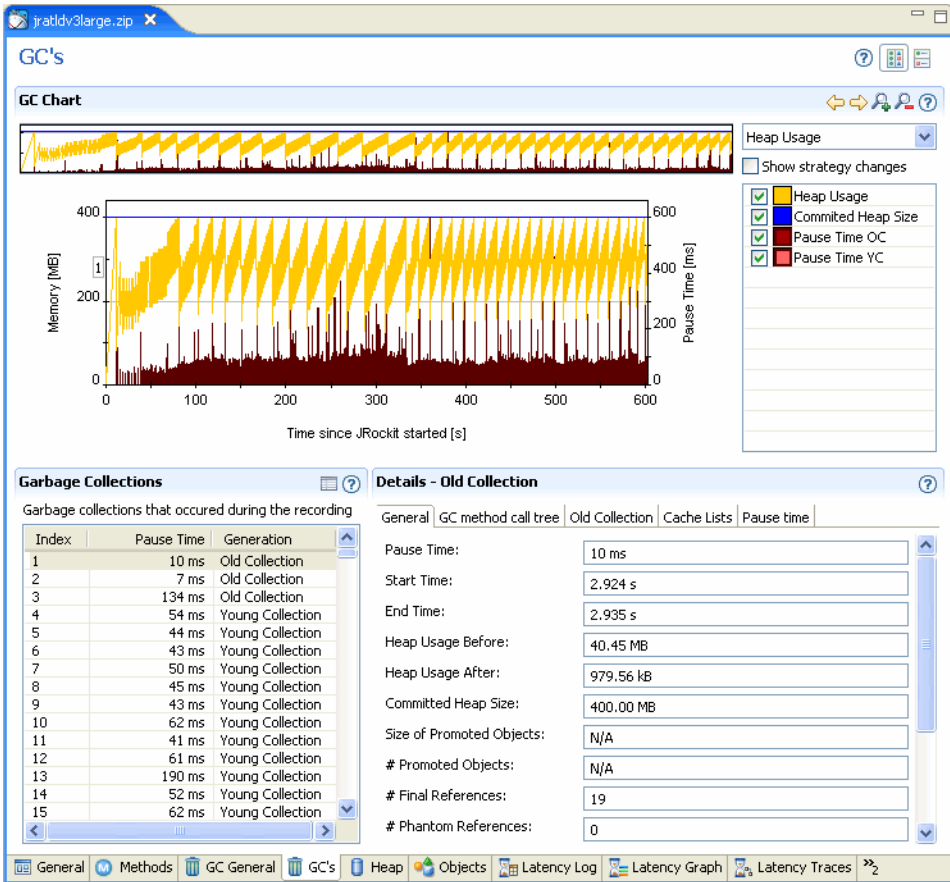
Figure 5-3 The JRockit Management Console



The JRockit Runtime Analyzer (JRA)

The JRockit Runtime Analyzer (see [Figure 5-4](#)) is an on-demand “flight recorder” that produces detailed recordings about the JVM and the application it is running. The recorded profile can later be analyzed off line, using the JRA. Recorded data includes profiling of methods and locks, as well as garbage collection statistics, optimization decisions, and event latencies.

Figure 5-4 The JRockit Runtime Analyzer



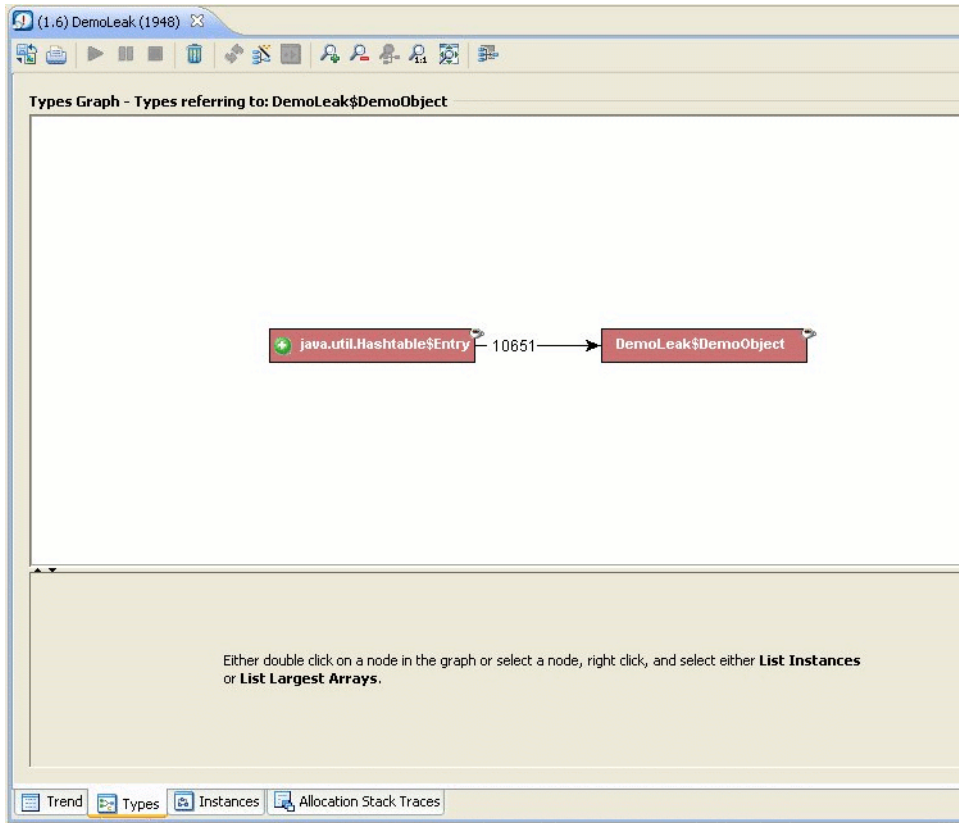
The JRockit Memory Leak Detector

Note: The Memory Leak Detector is not encrypted; therefore, you should **never** use it on a public network. An encryption protocol will be available in a future edition of this product.

The JRockit Memory Leak Detector (see [Figure 5-5](#)) is a tool for discovering and finding the cause for memory leaks in a Java application. The JRockit Memory Leak Detector’s trend analyzer discovers slow leaks, it shows detailed heap statistics (including referring types and instances to leaking objects), allocation sites, and it provides a quick drill down to the cause of

the memory leak. The Memory Leak Detector uses advanced graphical presentation techniques to make it easier to navigate and understand the sometimes complex information.

Figure 5-5 The JRockit Memory Leak Detector



Overview of JRockit Mission Control Client 3.1.0

Overview of JRockit Mission Control 2.0

The Oracle JRockit Mission Control 2.0 tools suite includes tools to monitor, manage, profile, and eliminate memory leaks in your Java application without introducing the performance overhead normally associated with these types of tools.

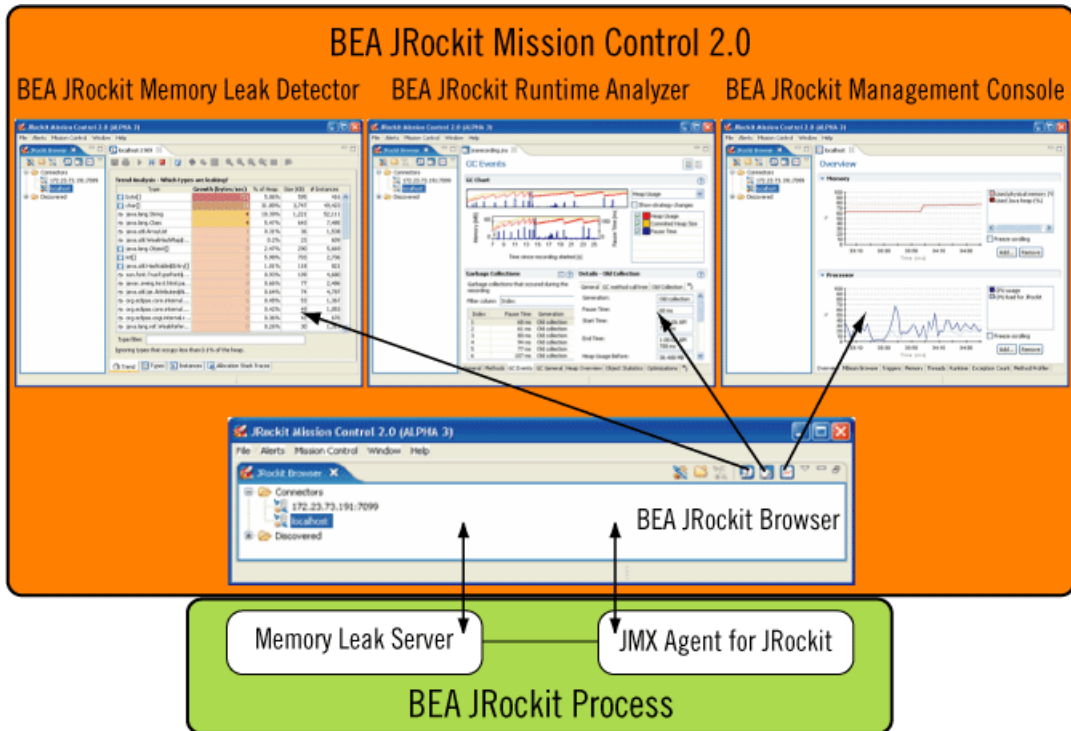
This section contains information on the following subjects:

- [Architectural Overview of JRockit Mission Control 2.0](#)
- [Starting JRockit Mission Control](#)
- [The JRockit Browser](#)
- [The JRockit Management Console](#)
- [The JRockit Runtime Analyzer \(JRA\)](#)
- [The JRockit Memory Leak Detector](#)

Architectural Overview of JRockit Mission Control 2.0

With the new Rich Client Platform (RCP) based JRockit Mission Control, you can launch the Memory Leak Detector, the JRockit Runtime Analyzer, and the JRockit Management Console from within the JRockit Mission Control (see [Figure 6-1](#)).

Figure 6-1 Architectural Overview of JRockit Mission Control 2.0



With the JMX Agent you have access to all MBeans deployed in the platform MBean server. From these MBeans, you can read attributes information, such as garbage collection pauses.

When a JRA recording is started from within JRockit Mission Control, it records the status of the JRockit JVM process for the time that you have specified and creates an XML file. This file is automatically opened in the JRockit Runtime Analyzer. Typical information that is recorded during a JRA recording is Java heap distribution, garbage collections, method optimizations, and method profiling information.

To find memory leaks in your Java application, you connect the JRockit Memory Leak Detector to the running JRockit JVM process. The Memory Leak Detector connects to the JMX (RMP) Agent that instructs to start a Memory Leak server where all further communication takes place.

Starting JRockit Mission Control

The JRockit Mission Control executable is located in `JROCKIT_HOME/bin`. If this directory is on your system path, you can start JRockit Mission Control by simply typing `jrmc` in a command (shell) prompt.

Otherwise, you have to type the full path to the executable file, as shown below:

```
JROCKIT_HOME/bin/jrmc.exe (Windows)
```

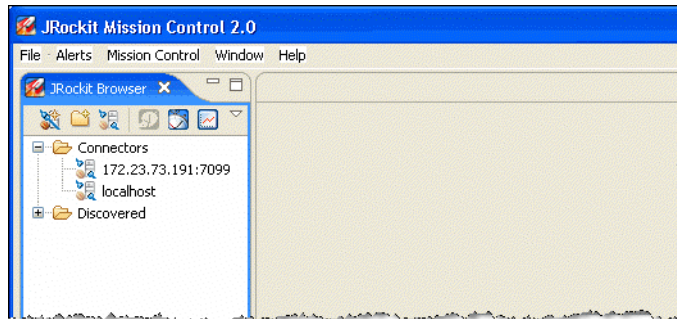
```
JROCKIT_HOME\bin\jrmc (Linux)
```

On Windows installations, you can start JRockit Mission Control from the **Start** menu.

The JRockit Browser

The JRockit Browser (see [Figure 6-2](#)) is new for the JRockit Mission Control 2.0 release. This tool allows you to set up and manage all running instances of JRockit JVM on your system. From the JRockit Browser you activate recordings, set up a tree view of different JRockit JVM instances to monitor, start other JRockit Mission Control tools, etc. Each JRockit JVM instance is referred to as a Connector.

Figure 6-2 The JRockit Browser

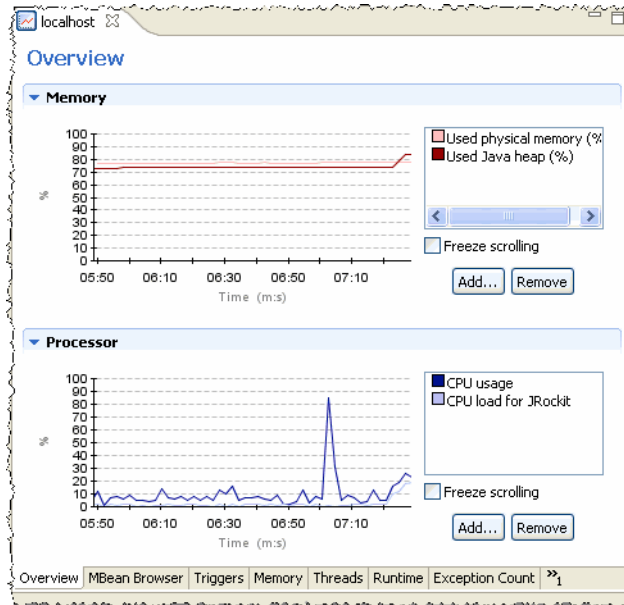


The JRockit Management Console

The JRockit Management Console (see [Figure 6-3](#)) is used to monitor a JRockit JVM instance. Several Management Consoles can be running concurrently side by side. The tool captures and presents live data about memory, CPU usage, and other runtime metrics. For the Management Console that is connected to Oracle JRockit JDK 5.0, information from any JMX MBean deployed in the JRockit JVM internal MBean server can be displayed as well. For a Console connected to Oracle JRockit JDK 1.4, RMP capabilities are exposed by a JMX proxy. JVM

management includes dynamic control over CPU affinity, garbage collection strategy, memory pool sizes, and more.

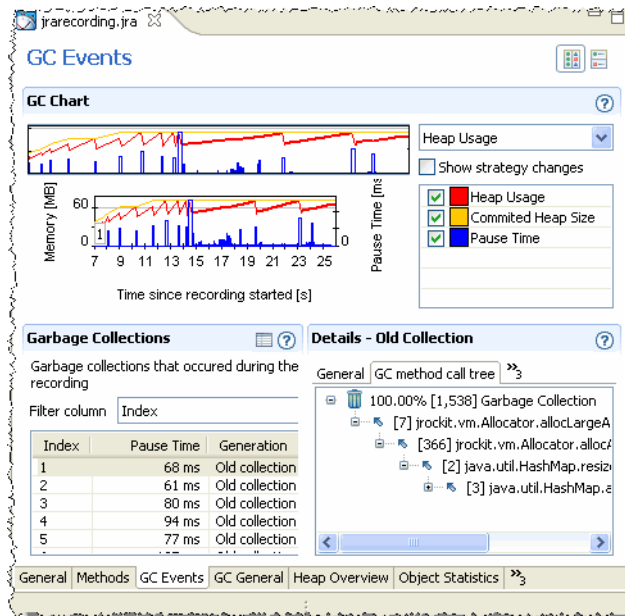
Figure 6-3 The JRockit Management Console



The JRockit Runtime Analyzer (JRA)

The JRockit Runtime Analyzer (see [Figure 6-4](#)) is an on-demand “flight recorder” that produces detailed recordings about the JVM and the application it is running. The recorded profile can later be analyzed off line, using the JRA tool. Recorded data includes profiling of methods and locks, as well as garbage collection statistics, optimization decisions, and object statistics.

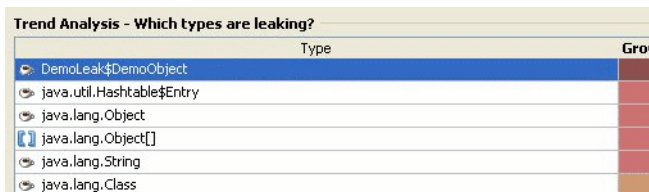
Figure 6-4 The JRockit Runtime Analyzer



The JRockit Memory Leak Detector

The JRockit Memory Leak Detector (see [Figure 6-5](#)) is a tool for discovering, and finding the cause for memory leaks in a Java application. The JRockit Memory Leak Detector's trend analyzer discovers slow leaks, it shows detailed heap statistics (including referring types and instances to leaking objects), allocation sites, and it provides a quick drill down to the cause of the memory leak. The Memory Leak Detector uses advanced graphical presentation techniques to make it easier to navigate and understand the sometimes complex information.

Figure 6-5 The JRockit Memory Leak Detector



The screenshot shows a window titled "Trend Analysis - Which types are leaking?". It contains a table with two columns: "Type" and "Gro". The table lists several Java classes, with "DemoLeak\$DemoObject" selected. The "Gro" column has a color gradient from dark red to light brown, indicating the severity or frequency of the leak.

Type	Gro
DemoLeak\$DemoObject	Dark Red
java.util.Hashtable\$Entry	Red
java.lang.Object	Red
java.lang.Object[]	Red
java.lang.String	Red
java.lang.Class	Light Brown

Overview of Oracle JRockit 1.0

The Oracle JRockit Mission Control 1.0 tools suite was introduced with Oracle JRockit JDK R26.0.0. These tools are run as standalone tools to the JRockit JVM.

This section contains information on the following subjects:

- [Architectural Overview of JRockit Mission Control 1.0](#)
- [The JRockit Management Console](#)
- [The JRockit Runtime Analyzer \(JRA\)](#)
- [The JRockit Memory Leak Detector](#)

Architectural Overview of JRockit Mission Control 1.0

JRockit Mission Control is available on Oracle JRockit JDK 1.4.2 (R26.2 and later) and JRockit JVM 5.0 (R26.0 and later), see [Figure 1-1](#). The difference between the two is the connection agent used by the JRockit Management Console and the JRockit Management Console user interface itself.

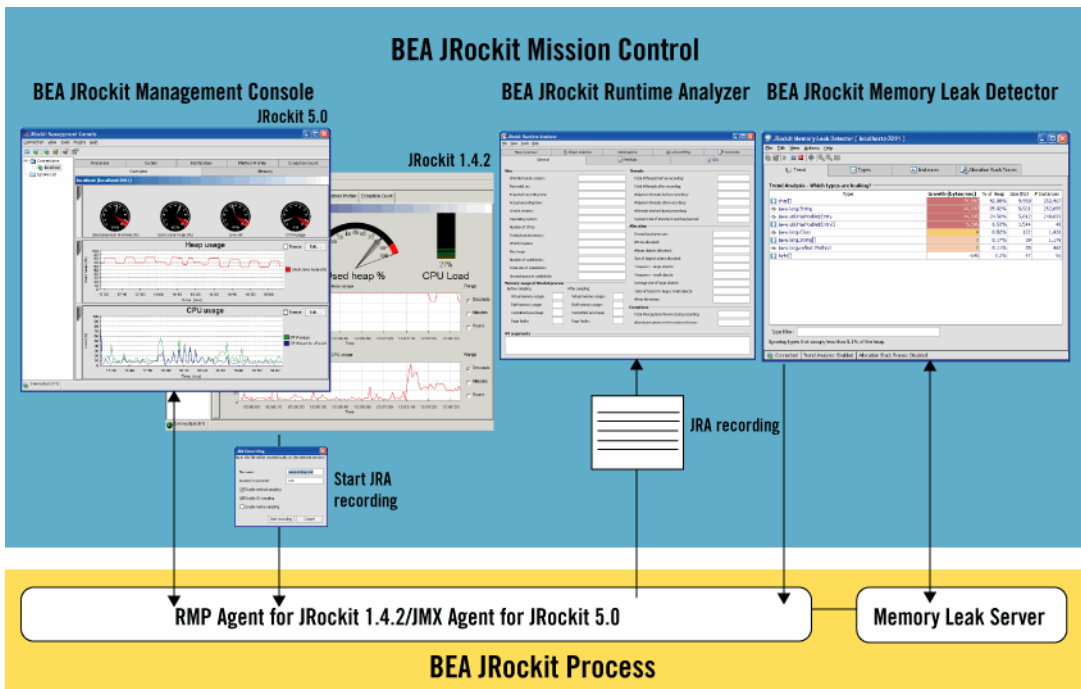
The RMP Agent (JRockit JDK 1.4.2) provides access, among other things, to live data about memory and CPU usage. With the addition of the JMX Agent (available with JRockit JDK 5.0) you will also get access to MBeans available to the platform MBean server. From these MBeans, you can read attributes information, such as garbage collection pauses.

When a JRA recording is started, for example, from the Management Console, it records the status of the JRockit JVM process for the time that you have specified. When the recording is completed, the information is saved to an XML file. This XML file can be viewed and analyzed

in the JRockit Runtime Analyzer tool. Typical information that is recorded during a JRA recording is Java heap distribution, garbage collections, and method optimizations.

To find memory leaks in your Java application, you connect the JRockit Memory Leak Detector to the running JRockit JVM process. The Memory Leak Detector connects to the JMX (RMP) Agent that instructs to start a Memory Leak server where all further communication takes place.

Figure 1-1 Architectural Overview of JRockit Mission Control 1.0



The JRockit Management Console

The JRockit Management Console is used to monitor and manage multiple (or single) JRockit JVM instances. It captures and presents live data about memory, CPU usage, and other runtime metrics. For the Management Console that is running on JRockit JDK 5.0, information from any JMX MBean deployed in the JRockit JVM internal MBean server (JMX Agent in Figure 1-1) can be displayed as well. JVM management includes dynamic control over CPU affinity, garbage collection strategy, memory pool sizes, and more.

The JRockit Runtime Analyzer (JRA)

The JRockit Runtime Analyzer (JRA) is an on-demand “flight recorder” that produces detailed recordings about the JVM and the application it is running. The recorded profile can later be analyzed off line, using the JRA tool. Recorded data includes profiling of methods and locks, as well as garbage collection statistics, optimization decisions, and object statistics.

The JRockit Memory Leak Detector

The JRockit Memory Leak Detector is a tool for discovering, and finding the cause for memory leaks in a Java application. The JRockit Memory Leak Detector’s trend analyzer discovers slow leaks, it shows detailed heap statistics (including referring types and instances to leaking objects), allocation sites, and it provides a quick drill down to the cause of the memory leak. The Memory Leak Detector uses advanced graphical presentation techniques to make it easier to navigate and understand the sometimes complex information.

Overview of Oracle JRockit 1.0

Oracle JRockit Mission Control Client Use Cases

This chapter demonstrates various ways Oracle JRockit Mission Control Client can be used to monitor and manage application running on the Oracle JRockit JVM. It includes the following use cases:

- [Analyzing System Behavior with the JRockit Management Console](#)
- [Analyzing System Problems with the JRockit Runtime Analyzer](#)
- [Detecting a Memory Leak](#)

Analyzing System Behavior with the JRockit Management Console

Marcus wants to monitor his application, DemoLeak which he's running on an instance of the JRockit JVM, to ensure that he has tuned it to provide the best possible performance. To do this, he will run the JRockit Management Console concurrent with the application run. The Management Console will provide realtime information about memory, CPU usage, and other runtime metrics. The Management Console is a multi-tabbed interface, each tab allowing him to monitor and/or manage an aspect of a running application. Which tabs his version of the Management Console uses depends on which Java plug-ins he has installed with the console. When fully-implemented, the console will include eight tabs and one menu, which map to seven plug-ins.

Getting Started

To get started, Marcus launches the JRockit Mission Control Client from the command prompt, by entering:

```
jrockit\bin\jrnc
```

While the JRockit Mission Control Client is starting up, he launches the DemoLeak application. At the command prompt, he enters:

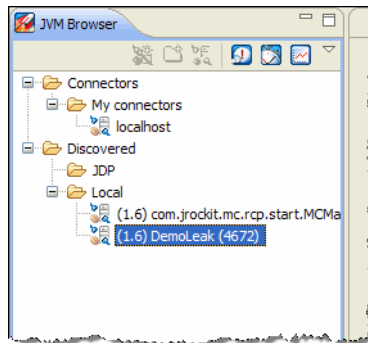
```
jrockit\bin\java DemoLeak
```

Next, he starts the Management Console with a local connection.

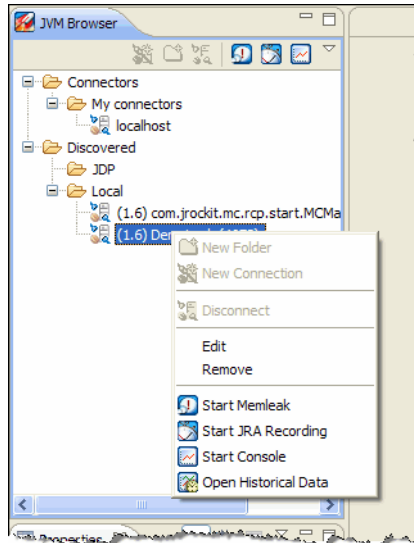
To launch the Management Console, Marcus does the following:

1. In the JRockit Browser, he locates the JRockit JVM instance to which he wants to connect. In this case, it will be the one running the DemoLeak class under **Discovered/Local**.

Figure 2-1 Locating the Appropriate JRockit JVM Instance



2. He right-clicks the mouse to open a context menu for the connection.

Figure 2-2 Context Menu for Selected JRockit JVM Instance

3. He selects **Start Console**.

After a few moments, the Management Console appears in the right panel of the JRockit Mission Control Client. Note that in Marcus's implementation of the JRockit Mission Control Client, he can see the following tabs:

- Overview
- MBean Browser
- Memory
- Threads
- Runtime
- Triggers
- Exception Count
- Method Profiler

Analyzing Memory Usage

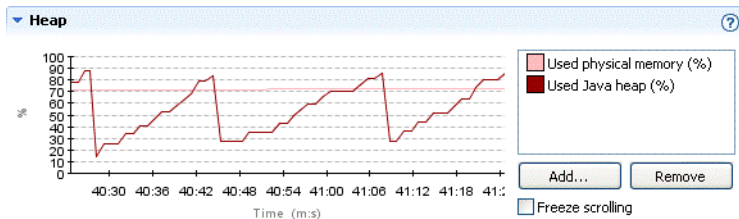
One way to spot problems with application performance is to see how it uses memory during runtime. To analyze how his application is using the memory available to it, Marcus will use the

Memory Tab. This tab focuses on heap usage, memory usage, and garbage collection schemes. The information provided on this tab can greatly assist Marcus in determining whether he has configured the JRockit JVM to provide optimal application performance.

To analyze memory usage, Marcus does the following:

1. First, he examines the Heap graph, which shows the used Java heap growing until it reaches 80% to 90% of the available heap before a garbage collection is triggered. At that point the graph falls back, indicating that new heap space is available. As the graph in Figure shows, this cycle repeats itself throughout the run.

Figure 2-3 Heap Graph



2. Next, he takes a look at the Memory Statistics and Garbage Collector panels, which show additional information about memory usage and the garbage collector, respectively. If necessary, Marcus can change some of the values from this tab; for example he could change the allocated Java heap size or the garbage collection strategy if he felt that those originally selected weren't allowing the application to run optimally.

Plotting Garbage Collection Times

Next, he decides to see the duration for each garbage collection. Overly long garbage collection times are a common cause of poor application performance. To see the duration of the garbage collections, Marcus can plot this information on the Heap graph.

The graphs shown in the various tabs are all preconfigured with a few useful default attributes, but any numerical attribute from any MBean can be added. In addition to the standard MBeans in J2SE 5.0 and the JRockit JVM specific MBeans, JRockit Mission Control itself provides so called synthetic MBeans that derives attributes from multiple other attributes. One such attribute is the garbage collection times

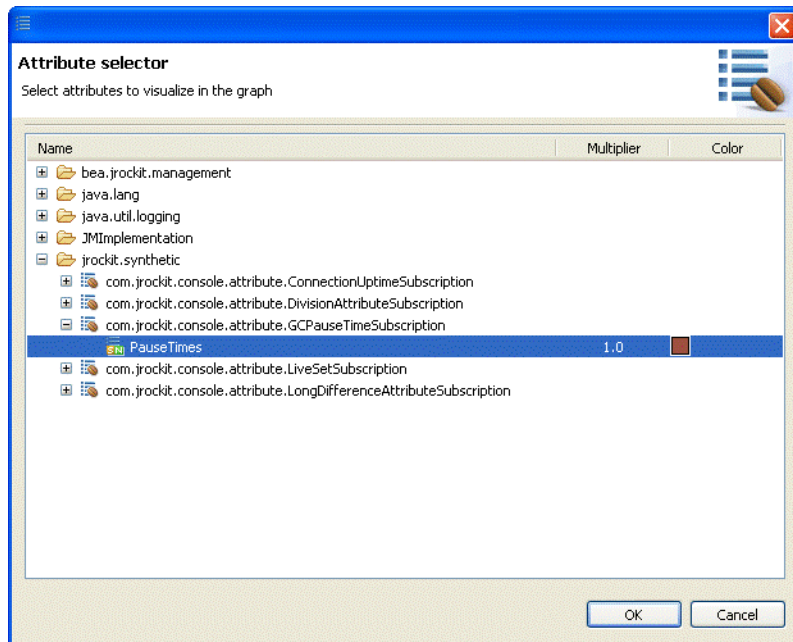
Note: The attribute for garbage collection durations is called PauseTimes even though the Java application isn't necessarily paused during the whole garbage collection. When a concurrent garbage collector is in use, the garbage collector runs concurrently with the Java application for the most part of the garbage collection duration. The misleading

naming of the attribute is a known issue and will be fixed in upcoming releases. The correct name of the attribute would be Duration.

To do this, Marcus does the following:

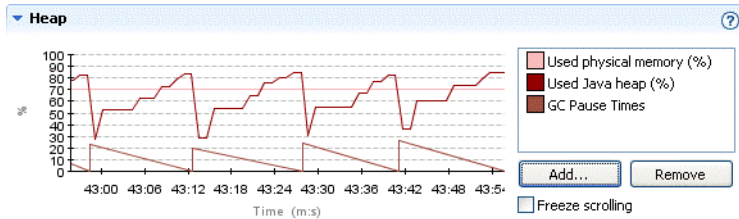
1. In the Memory tab, he clicks **Add...** (to the right of the Heap graph).
The Attribute Selector appears.
2. He drills down to the PauseTimes attribute, as shown in [Figure 2-4](#), select it and clicks **OK**.

Figure 2-4 Selecting an Attribute to Add to the Heap Graph



The new attribute should now be shown in the Heap graph. This synthetic attribute is a somewhat special in that it only shows values just before and after a garbage collection, causing the triangular-shaped plot, as shown in [Figure 2-5](#). The value is shown in milliseconds.

Figure 2-5 Heap Graph with Pause Time Plot Added



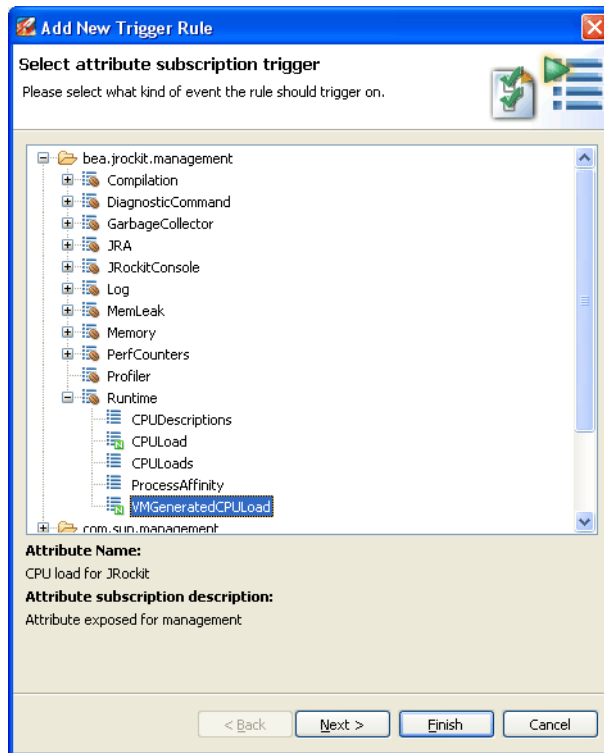
Setting an Alert Trigger

In his search for bottlenecks in the system, Marcus looks at the CPU load graph and notices that the CPU load for the JVM sometimes hits the roof. Marcus would like to know how often this happens for a longer period of time. Instead of staying and watching the CPU graph continuously he sets an alert trigger to alert him whenever the CPU load generated by the VM is high for a longer period of time.

To set the alert, Marcus does the following:

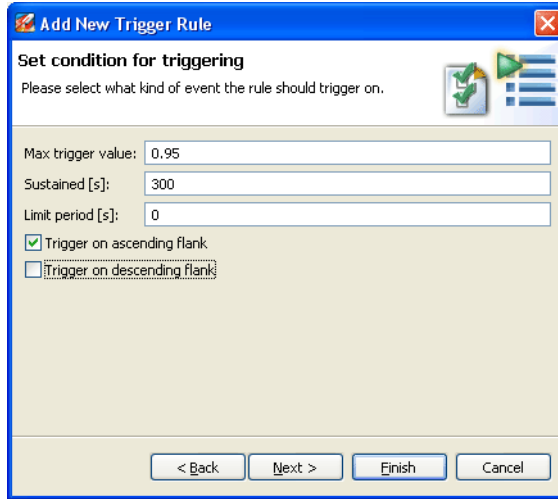
1. Marcus goes to the Triggers tab and clicks **Add...** (under Trigger Rules).
The Add New Trigger Rule wizard appears.
2. He drills down and selects the VMGeneratedCPULoad as shown in [Figure 2-6](#). He then clicks **Next >**.

Figure 2-6 Selecting an Attribute to Trigger On



3. He enters the conditions as shown in [Figure 2-7](#).

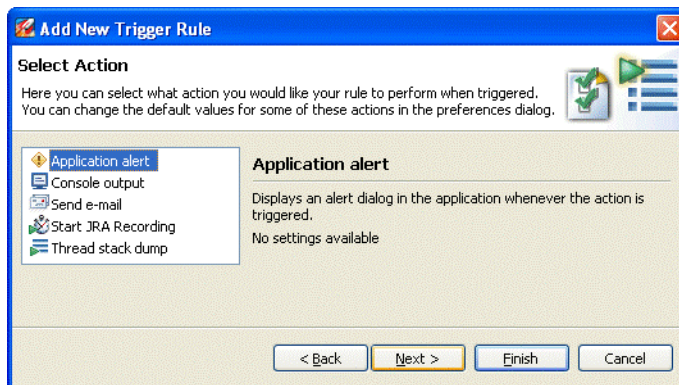
Figure 2-7 Setting Conditions for Triggering



Since the CPU load value ranges from 0-1, Marcus sets the **Max trigger value** to 0.95. Marcus wants to be alerted when the CPU usage is high for at least five minutes and sets **Sustained [s]** to 300. He sets the **Limit period [s]** to 10 to prevent triggers less than 10 seconds apart.

- 4. He clicks **Next >**.
- 5. The Add New Trigger Rule: Select Action dialog box appears (Figure 2-8). Marcus selects **Application alert** and clicks **Next >**.

Figure 2-8 Selecting the Trigger Action



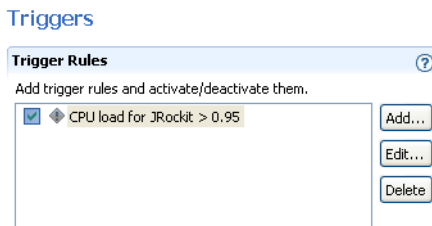
6. Marcus skips the optional constraints of when to arm the trigger by clicking **Next >**.
7. Marcus names the rule as shown in [Figure 2-9](#) (CPU load for JRockit > 0.95) and clicks **Finish**.

Figure 2-9 Naming the Rule



8. Marcus then activates the new rule by checking the box next to the rule name.

Figure 2-10 Trigger Selected



9. To verify that he is getting useful data, Marcus then returns to the Memory tab and checks the CPU activity. He notices that the Trigger Alerts dialog box doesn't appear, so he edits the rule by going back to the Triggers tab, selecting the rule and lowering the **Max trigger value** to 0.90 or so under **Trigger Condition**.
10. Since Marcus doesn't want the Triggers Alert dialog box to appear every time an event is triggered, he will uncheck **Show dialog on alerts** to prevent this from happening (he can display the dialog from the Window menu whenever he wants).

Profiling Methods Online by Using the Console

Next, Marcus wants to see how many times and for how long some specific methods have run, a process called method profiling. JRockit Mission Control has two tools for profiling methods:

- To run create a runtime analysis with the JRockit Runtime Analyzer (JRA), which we will demonstrate in [Analyzing System Problems with the JRockit Runtime Analyzer](#). While this is best way to find out which methods are most likely affecting performance, it is also the more complex tool to run.
- By using the Method Profiler tab in the Management Console. This tool provides efficient and detailed method profiling while requiring a minimal amount of overhead and system intrusion. It also allows you to profile an application for which you are already collecting and viewing other information on the console.

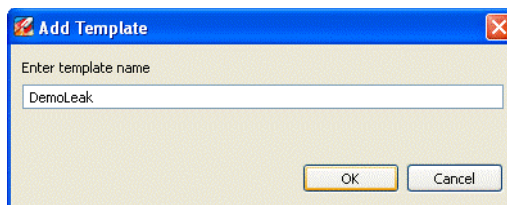
To profile methods by using the Method Profiler tab, Marcus does the following

1. First, he needs to create a method profiling template by going to the Method Profiler tab and click **Add...** in the Templates panel.

The Add Template dialog box appears.

2. He enters a name for the new template in the Add Template dialog box, as shown in [Figure 2-11](#).

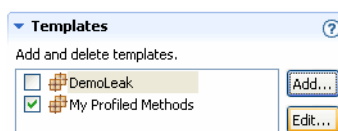
Figure 2-11 Template Name Added



3. He then clicks **OK**.

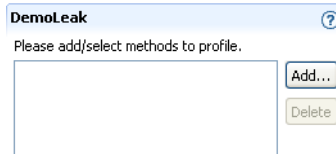
The dialog box closes and the new template is added to the list ([Figure 2-12](#)).

Figure 2-12 DemoLeak Template Added



4. He enables the new DemoLeak template by checking the box in front of the name, then selects the template and clicks **Add...** in the DemoLeak panel (Figure 2-13).

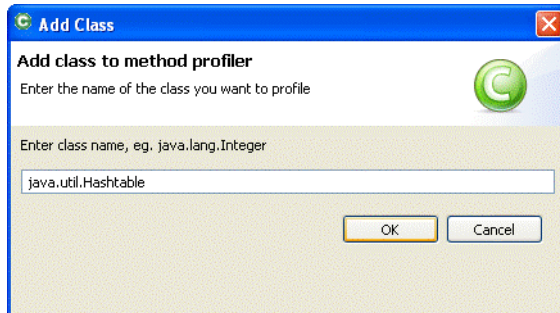
Figure 2-13 DemoLeak Panel



Add class to method profiler dialog box appears.

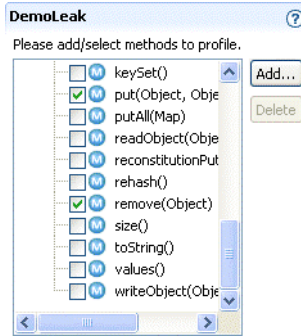
5. In the Add class to method profiler dialog box, he enters `java.util.Hashtable`, as shown in Figure 2-14

Figure 2-14 Adding java.util.Hashtable Class



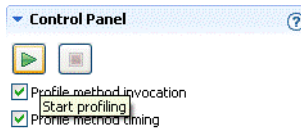
6. He clicks **OK**.
7. In the DemoLeak panel, he expands the `java.util.Hashtable` class, scrolls down and checks the boxes in front of the `put(Object, Object)` and `remove(Object)` methods, as shown in Figure 2-15.

Figure 2-15 java.util.Hashtable Methods Selected



8. He then starts profiling by clicking the green play button in the Control Panel (Figure 2-16).

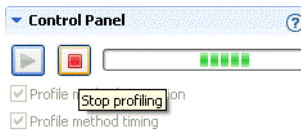
Figure 2-16 Start Profiling Button



9. To interpret the results, he examines the Profiling Information panel. He notes how the number of invocations of Hashtable.put(Object, Object) grows slightly faster than the number of invocations of Hashtable.remove(Object).

10. Marcus then stops profiling by clicking the red stop button in the Control Panel (Figure 2-17).

Figure 2-17 Stop Profiling Button



Analyzing System Problems with the JRockit Runtime Analyzer

Fiona is not happy with how the application DemoLeak is performing. She is particularly concerned about the way her application performs the longer it runs. For example, while the application works fine early in its run, after a while, it starts reporting the wrong results and throwing exceptions where it shouldn't. She also notices that eventually, it hangs at roughly the

same time every time she runs it. To assess what the problem is, Fiona decides to create a runtime analysis by use the JRockit Runtime Analyzer (JRA).

The JRA is an on-demand “flight recorder” that produces detailed recordings about the JVM and the application it is running. The recorded profile can later be analyzed off line, using the JRA Tool. Recorded data includes profiling of methods and locks, as well as garbage collection statistics, optimization decisions, and, in JRockit Mission Control 3.1.0, latency analysis.

Getting Started

To start the diagnostics process, Fiona does the following:

1. She starts the JRockit Mission Control Client from the command line by typing:

```
jrockit\bin\jrnc
```

2. While the JRockit Mission Control Client starts, Fiona starts the DemoLeak by entering:

```
jrockit\bin\java DemoLeak
```

Creating the Recording

Next, Fiona creates a JRA recording from a local connection. To do so, Fiona does the following:

1. Launches the JRockit Mission Control Client and locates in the JRockit Browser the JRockit JVM instance to which she wants to connect. This should be the one running the DemoLeak class under Discovered/Local.
2. Right-clicks the mouse to display a context menu for the selected connection.
3. Selects **Start JRA Recording** to launch the Start JRA Recording wizard.
4. Selects the connection to the JRockit JVM instance on which she wants to start the recording.
5. Selects filename and directory and types a descriptive name for the recording in the **Local filename** field. Note that the JRA recording file is created in the current directory of the JRockit JVM process, unless Fiona specifies a different path. If an old file already exists, it will be overwritten by the new recording.
6. Enters the desired length of the recording (in seconds) in **Recording time**.

Note: If Fiona sets a recording length that is too short, for example, less than 30 seconds, she will probably not get enough sample data for the recording to be meaningful.
7. Selects the sampling options, as described in [Table 2-1](#):

Table 2-1 Selected Sampling Options

Sampling Option	Description
Record samples of methods	Records samples of methods
Use gc sampling	Records garbage collection events
Use native sampling	Records samples of native code
Compress recording	Compresses recording to a zip file
Selected JRockits	Shows the JRockit JVM instance from which she will create her recording

8. Clicks **Finish**.

The JRA recording progress window appears. When the recording is finished, it loads in the JRA. Fiona will now look at the JRA Recording.

Looking at the Recording

Next, Fiona will use the JRockit Mission Control Client to view the JRA recording. First, she opens the General tab by doing the following:

1. In the JRockit Mission Control Client, clicking **File > Open file > Open JRA Recording**.
2. Locating and selecting the recorded file and clicking **Open**.
3. Clicking **OK**.

The JRA General tab for that recorded file now opens, allowing Fiona to view the data in the recording. The General tab contains information on the JVM, your system and your application. It is divided into the panels described in [Table 2-2](#):

Table 2-2 General Tab Sections

Data Field	Description
General Information	Contains all general information about the JVM, operating system, recording time, and so on.
Memory Usage	Contains information on how the JRockit JVM is using the memory.

Table 2-2 General Tab Sections

Data Field	Description
VM Arguments	Lists all startup options that were used.
Allocation	Contains information on how your application allocates memory on the Java heap.
Threads	Contains information on thread usage.
Exceptions	Contains exceptions related information.

By looking at this tab, Fiona is able to verify which version of the JVM she was running. She can also see that large object were allocated at a rate, or “frequency”, of 22.153 MB per second while small objects were allocated at a significantly faster rate of 261.983 MB per second.

Examining the Methods Tab

Next, Fiona will look at the Methods tab. The Method tab lists the top hot methods with their predecessors and successors during the recording. The Methods tab is divided into the following panels described in [Table 2-3](#):

Table 2-3 Methods Tab Panels

Field	Description
Top Hot Methods	A listing of the top hot methods. A hot method is defined as the methods where the JVM spends most of its time during application execution. Being “hot” might indicate that a specific method is causing system problems.
Predecessors	A listing of all methods called prior to calling the method Fiona selected in the Top Hot Methods list. This information can be helpful in determining if some aspect of a certain method is complicit is poor system performance. If Fiona selects too many methods, no information will appear in this section.
Successors	A listing of all methods called after the calling method that Fiona selected in the Top Hot Methods list. This information can be helpful in determining if some aspect of a certain method is complicit is poor system performance. If Fiona selects too many methods, no information will appear in this section.

Examining the Top Hot Methods

The method sampling in the JRockit JVM is based on CPU sampling. The Top Hot Methods section lists all methods sampled during the recording and sorts them with the most sampled methods first, as shown in Figure 2-18.

Figure 2-18 Top Hot Methods

The methods where JRockit spent the most time executing. Total number of samples are 194.

Filter column: Method

Method	%	#Samples
java.util.Hashtable.put(Object, Object)	38.66%	75
java.util.Hashtable.remove(Object)	9.28%	18
DemoLeak\$DemoThread.put(int)	6.70%	13
DemoLeak\$DemoThread.remove(int)	4.12%	8
java.util.Hashtable.rehash()	3.09%	6
DemoLeak\$DemoThread.run()	3.09%	6
java.util.Hashtable.put(Object, Object)	2.06%	4
jrockit.vm.Locks.monitorEnter(Object)	2.06%	4
jrockit.vm.Allocator.innerAllocate(int, int, int, boo...	2.06%	4
jvm.dll#_qBitSetClear	2.06%	4
jrockit.vm.Allocator.innerAllocate(int, int, int, boo...	2.06%	4
DemoLeak\$DemoObject.equals(Object)	2.06%	4
DemoLeak\$DemoObject.hashCode()	1.55%	3
java.util.Hashtable.remove(Object)	1.55%	3
java.util.Hashtable.rehash()	1.55%	3
jrockit.vm.Locks.lockThin(Object, boolean)	1.55%	3
DemoLeak\$DemoObject.equals(Object)	1.03%	2

Note: If Fiona enabled native sampling during the recording, she would see symbols with a pound sign, such as `jvm.dll#_qBitSetClear`. These denote functions in native libraries such as the JVM itself or various operating system libraries.

By looking at the list of top hot methods, Fiona sees that the three hottest methods are:

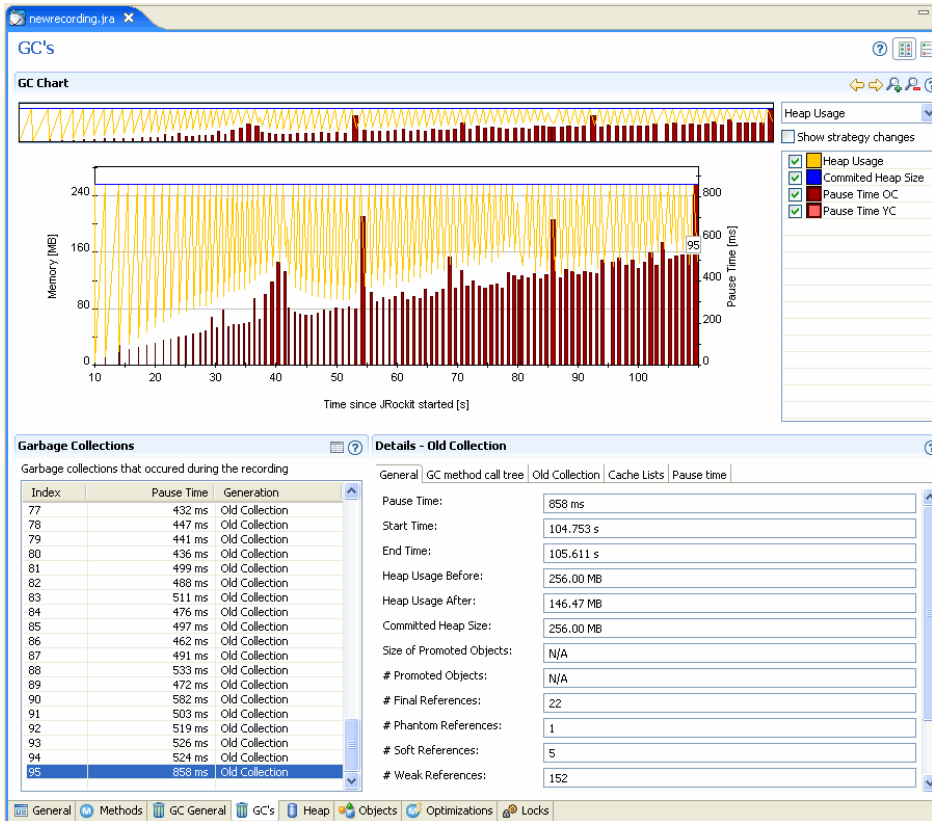
- `java.util.Hashtable.put (Object)`
- `java.util.Hashtable.remove (Object , Object)`
- `DemoLeak$DemoThread.put (int)`

Starting with this information, Fiona has a good idea of where to start looking for possible areas of concern. Fiona knows that the hottest methods are those that are sampled most often. In some situations, the number of samplings in the hottest methods will dwarf those of the less-hot methods. Hot methods are a good indicator of performance problems, especially memory leaks, because the high amount of sampling affects how much time the JVM has been executing the specific method.

Examine Garbage Collection Events

Next Fiona examines the GC's tab (Figure 2-19) to better understand system behavior and garbage collection performance during runtime.

Figure 2-19 GC's Tab

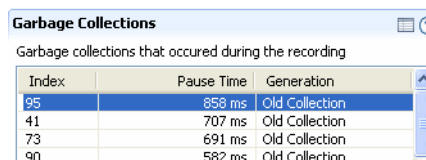


This tab is divided into the six panels described in Table 2-4

Table 2-4 GC Events Tab Panels

Panel	Description
GC Events Overview timeline	This timeline shows the entire recording based on when the recording is initially started. Fiona uses this information to refocus the Heap Usage graph.
Heap Usage graph	This graph shows heap usage compared to pause times and how that varies during the recording. When Fiona selects a specific area in the GC Events Overview, she only sees that section of the recording. She can change the graph content in the Heap Usage drop-down list (marked 6 in Figure 2-19) to get a graphical view of the references and finalizers after each old collection.
Garbage Collections events	This list shows all garbage collection events that occurred during the recording. When she clicks a specific event, Fiona will see a flag in the Heap Usage graph for that particular event.
Details	This panel contains all the details about the specific garbage collection round. When Fiona selects a garbage collection in the Garbage Collection list, the tabs in the Details section change depending on whether or not she selected an old collection or a young collection.
Chart Configuration	This panel lets Fiona change the appearance on the active chart.
Heap Usage	Fiona uses this list to toggle the view on the Heap Usage chart to view References and finalizers. It shows different types of reference counts after each collection.

Looking at the data in the Garbage Collections panel ([Figure 2-20](#)), Fiona sees that the three longest garbage collection pause times are indexed 95 (856 ms), 41 (707 ms), and 73 (691 ms).

Figure 2-20 Garbage Collection Panel


The screenshot shows a window titled "Garbage Collections" with a subtitle "Garbage collections that occurred during the recording". It contains a table with three columns: Index, Pause Time, and Generation. The data is as follows:

Index	Pause Time	Generation
95	856 ms	Old Collection
41	707 ms	Old Collection
73	691 ms	Old Collection
91	582 ms	Old Collection

This data implied that, as processing continued on her application, garbage collections were taking longer. Fiona now has additional evidence to help diagnose what might be causing her

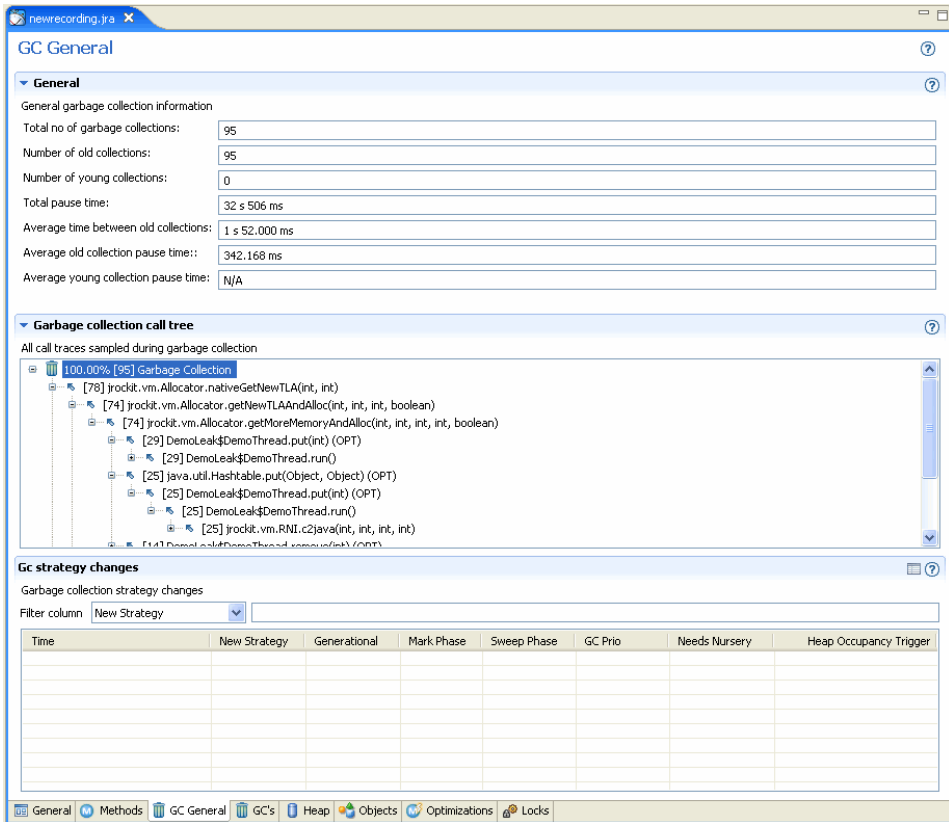
application's performance to deteriorate. She sees that garbage collection times are increasing, particularly later during runtime, and that the garbage collections are freeing less space on the heap. She can, with some confidence, predict that she is experiencing a memory leak.

Note: In this example, a memory leak is revealed to Fiona fairly quickly and she finds it because the evidence is obviously pointing in that direction. In most cases, a memory leak will reveal itself much more slowly and probably wouldn't be obvious on the GC's tab. Instead, a user would get better results by using the JRockit Memory Leak Detector, as described in [Detecting a Memory Leak](#).

Examine the GC General Tab

Fiona can gain more insight into how garbage collection activity might be indicating a memory leak by looking at the GC General tab ([Figure 2-21](#)).

Figure 2-21 GC General Tab



This tab is divided into three panels that provide information about the garbage collection at a glance. This tab is divided into the panels described in [Table 2-5](#):

Table 2-5 GC General Tab

Panel	Description
General	This section shows overall statistics about the garbage collections during the entire JRA recording.

Table 2-5 GC General Tab

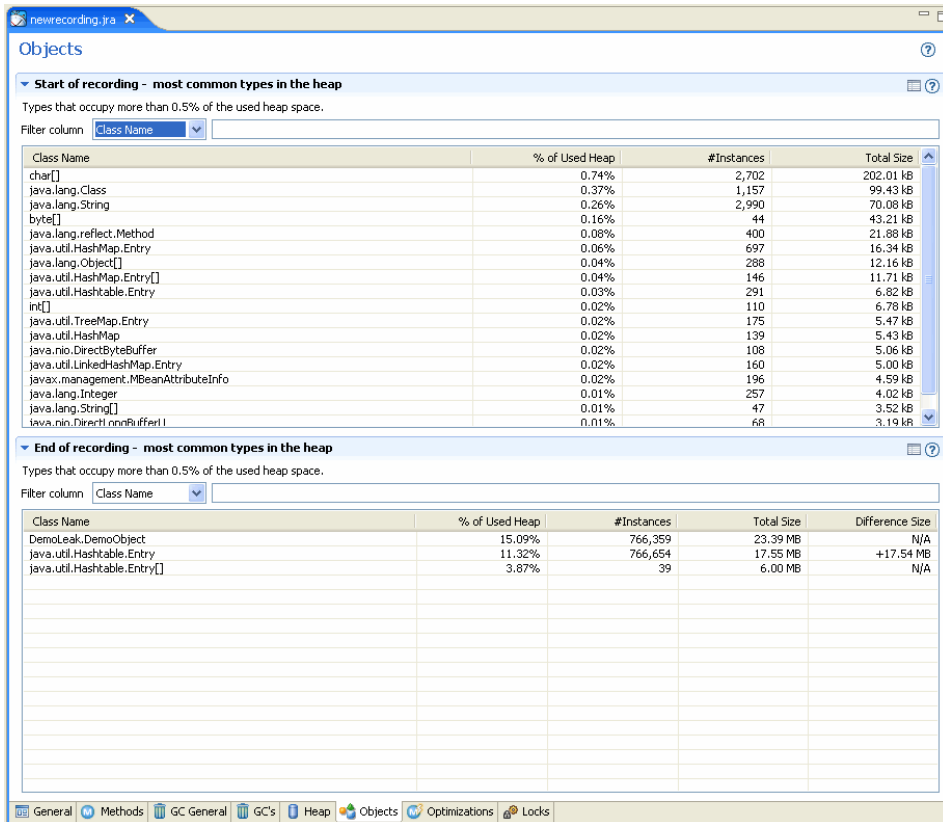
Panel	Description
Garbage Collection Call Tree	This section is a collection of all call traces that were sampled for all garbage collections for the JRA recording.
GC Strategy Changes	This section lists when a garbage collection strategy change took place and how it changed.

Fiona expands the stack tree down to user code and sees that many allocations are from the `hashtable` type, which indicates that this type is allocation intense. Reducing the allocation of this type would probably reduce the pressure on the memory management system.

Compare Object Statistics

Next, Fiona decides that it would be helpful to compare object statistics collected at the beginning of the recording to those collected after the recording. At the beginning and at the 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 of which the total number of instances occupy the most memory. The results are shown on the Objects tab ([Figure 2-22](#)).

Figure 2-22 Objects Tab



The Object Statistics tab is divided into the panels described in Table 2-6.

Table 2-6 Object Statistics Tab

Panel	Description
Start of Recording	This section lists the most common types on the heap at the beginning of the recording.
End of Recording	This section lists the most common types on the heap at the end of the recording.

Fiona can again see that `hashtable` shows the most dramatic growth and is consuming the greatest amount of memory on the heap. This again is a strong indication of not only a memory leak but that said leak involves the `hashtable` object.

Examine Lock Profiling Information

Fiona then checks the lock statistics for clues to performance bottlenecks involving locks. She opens the Locks tab ([Figure 2-23](#)) to investigate this information for both her application and the specific JRockit JVM instance.

Figure 2-23 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	Fat Cont.	Fat Uncont.	Fat Recursive	Thin Uncont.	Thin Cont.	Thin Recursive	Fat Sleep
java.security.BasicPermissionCollection	0	0	0	1	0	0	0
java.io.FilePermissionCollection	0	0	0	1	0	0	0
java.security.Permissions	0	0	0	2	0	0	0
sun.misc.URLClassPath\$FileLoader\$1	0	0	0	4	0	0	0
sun.misc.URLClassPath	0	0	0	12	0	0	0
sun.net.www.protocol.file.Handler	0	0	0	5	0	1	0
java.net.URL	0	0	0	8	0	0	0
java.io.ExpiringCache	0	0	0	2	0	0	0
java.util.Vector	0	0	0	7	0	0	0
java.util.Stack	0	0	0	2	0	2	0
java.util.Hashtable	0	1,568	0	307,465,270	52	0	0
java.util.HashMap	0	0	0	3	0	0	0
java.lang.StringBuffer	0	0	0	128	0	0	0
java.lang.ref.ReferenceQueue\$Lock	0	0	0	5	0	0	0
java.lang.ThreadGroup	0	0	0	7	0	0	0
sun.misc.Launcher\$AppClassLoader	0	0	0	5	1	21	0
sun.misc.Launcher\$ExtClassLoader	0	0	0	6	0	0	0
jrockit.memory.SoftCache\$ValueCell	0	0	0	5	0	0	0

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

Filter column:

Lock Name	Times Acquired	Times Contended	Times Try Failed
Thread System: Thread Table (0x00621D44)	100,229	102	0
GC: Wait For Action (0x005D8988)	404	89	0
GC: Start Action (0x005D69A8)	404	79	0
GC: Heap (0x005D8948)	2,618	36	0
GC: Finalize Notification (0x005DCC50)	202	12	0
CG Thread Worklist (0x005D90F0)	50	4	0
Periodic Task Lock (0x005DC858)	1,295,287	3	0
GC: Wait For OC (0x005D8868)	104	0	0
JRA Samples Buffer Lock (0x005D6B94)	197	0	0
GC: Non empty work pool (0x005C9CCD)	2,513	0	0
GC: Return FreeVec (0x005DCC90)	107	0	0
GC: Evacuation Finished (0x005D88A8)	156	0	0
GC: Task (0x005D88E8)	12,507	0	0
MemLeak Id Table (0x005D8C7C)	0	0	0
JRA Symbol Lock (0x005D8BD8)	303	0	0
GC: Promotion (0x005D8928)	0	0	0
GC: Scan (0x005D8968)	101	0	0
TypeGraph (0x00621FA8)	15	0	0

General Methods GC General GC's Heap Objects Optimizations Locks

The Lock Profiling tab is divided into the panels defined in [Table 2-7](#).

Table 2-7 Lock Profiling Tab Panels

Panel	Description
Java Locks	This section lists all locks in the application.
Native Locks	This section lists all JVM internal locks.

By looking at the Java Locks panel, Fiona can see immediately that the `hashtable` type has taken over 300 million uncontended locks, compared to a relative few for other objects. While this information does not point directly to a memory leak, it is indicative of poor performance.

Since the lock is mostly uncontended, Fiona could optimize her application by switching to an unsynchronized data structure such as `hashmap` and provide synchronization only for the few cases where contention may occur.

Detecting a Memory Leak

Since Fiona determined that a memory leak is causing her application to run poorly, she can take advantage of the JRockit Memory Leak Detector to confirm her suspicions and begin corrective action. A memory leak occurs when a program fails to release memory that is no longer needed. The term is actually a misnomer, since memory is not physically lost from the computer. Rather, memory is allocated to a program, and that program subsequently loses the ability to access it due to program logic flaws.

Getting Started

To start the memory leak detection process, Fiona does the following:

Note: This procedure assumes that the application was stopped after the JRA recording was completed. Had Fiona not stopped the application, she would be able to skip [step 1](#) and [step 2](#)

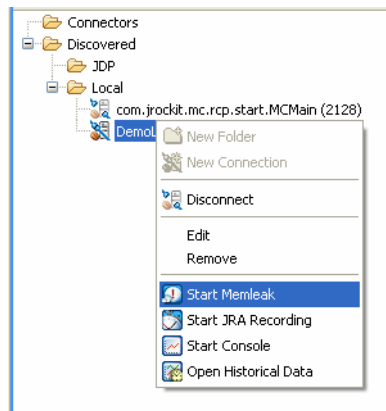
1. She starts the application by entering, at the command line:

```
java DemoLeak
```

2. While the application starts, she creates a connection to the server on which the application is running.
3. Next, she starts the Memory Leak Detector by doing the following:

- a. Right-clicking a Oracle JRockit JVM instance in the JRockit Browser to open a context menu.
- b. Selecting Start Memleak (Figure 2-24).

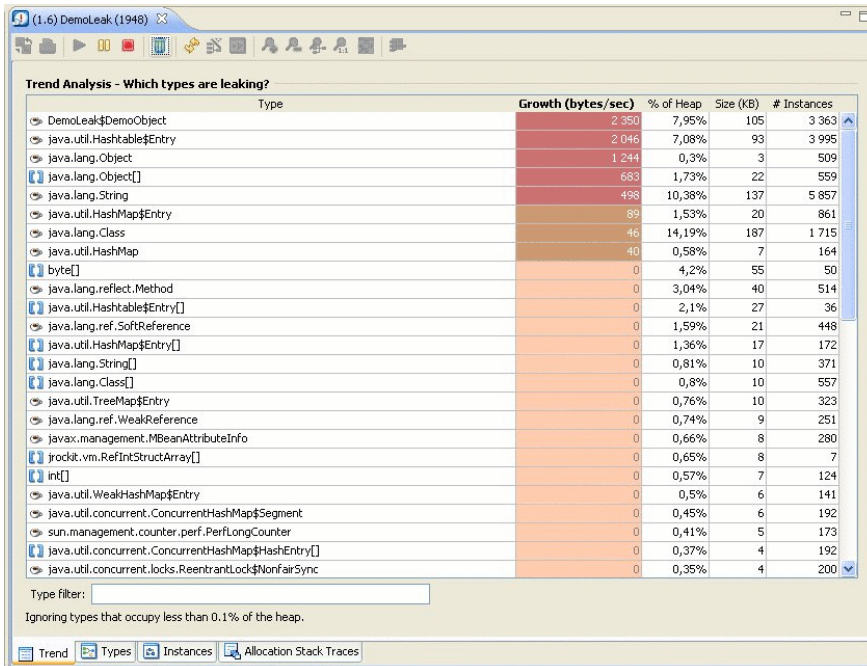
Figure 2-24 Starting The Memory Leak Detector from a Context Menu



Analyze the Java Application

Fiona starts the analysis from the Trend tab (Figure 2-25), which should open when she launches the Memory Leak Detector.

Figure 2-25 Trend Tab



Tip: The trend analysis should be running by default. If it is not running, you can start it by clicking the start symbol among the trend analysis buttons (Figure 2-27).

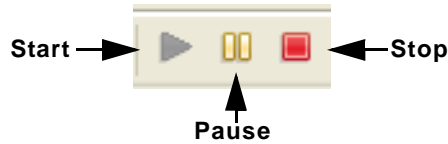
The trend analysis page shows Fiona the statistics on memory usage trends for the object types within the application. The JVM collects this data during garbage collections, which means that at least two garbage collections must be done before any trends are shown.

Figure 2-26 Garbage Collection Button



1. In order to speed up the process, Fiona clicks on the Garbage Collection button (Figure 2-26) a couple of times to start some garbage collections.

Figure 2-27 Trends Analysis Buttons



2. Fiona then pauses the trend analysis by clicking the pause symbol among the trend analysis buttons (Figure 2-27).

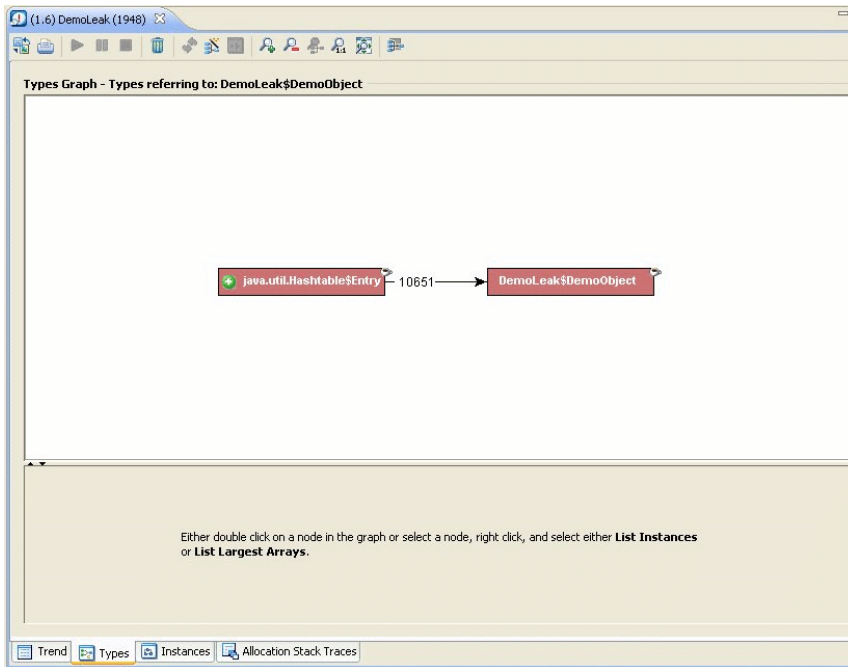
Figure 2-28 Show Referring Types

Trend Analysis - Which types are leaking?					
Type	Growth (bytes/sec)	% of Heap	Size (KB)	# Instances	
Demol.leak\$DemoObject	172	8,52%	137	5 872	8 590
java.util.Hashtable\$Entry					0 122
java.lang.Object					383
java.lang.Object[]					558
java.lang.String	15	11,61%	187	1 715	
java.lang.Class					

Fiona finds that a class named DemoObject shows the largest growth.

3. Fiona right-clicks the DemoObject class and selects “Show Referring Types” in the drop-down menu, as seen in Figure 2-28.

Figure 2-29 Types Tab

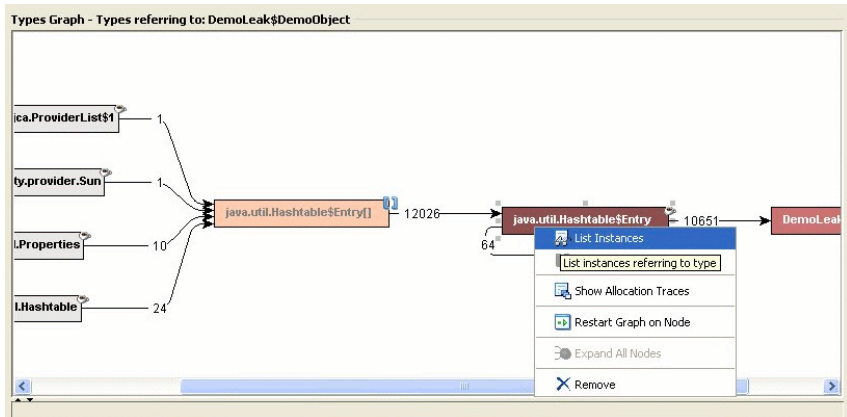


This opens the Types tab (Figure 2-29). Here Fiona can see that the DemoObjects are stored in hashtable entries.

4. Fiona clicks on the plus sign on the `java.util.Hashtable$Entry` node to expand the graph to show types referring to the hashtable entries.
5. Fiona continues expanding the graph to the left until she finds nodes in gray that do not show any growth trend.

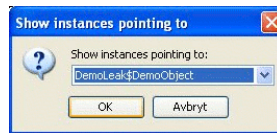
Now Fiona can see that the DemoObjects are indeed stored in hashtable entries. The next step for Fiona is to find out more about the instances holding on to the hashtable entries containing these DemoObjects.

Figure 2-30 List Instances



6. Fiona right-clicks the node that refers to the DemoObjects and selects List Instances in the drop-down menu, as seen in Figure 2-30.

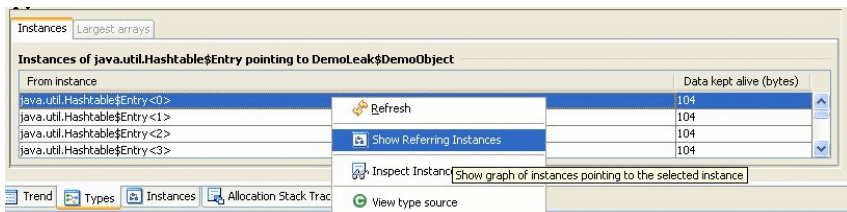
Figure 2-31 Show Instances Pointing To



Not all hashtable entries in the application point to the same type of objects, so Fiona gets a popup that asks her to select the type of references she is interested in.

7. Fiona selects DemoObject in the popup (Figure 2-31), since this is the type that she is interested in, and clicks Ok.

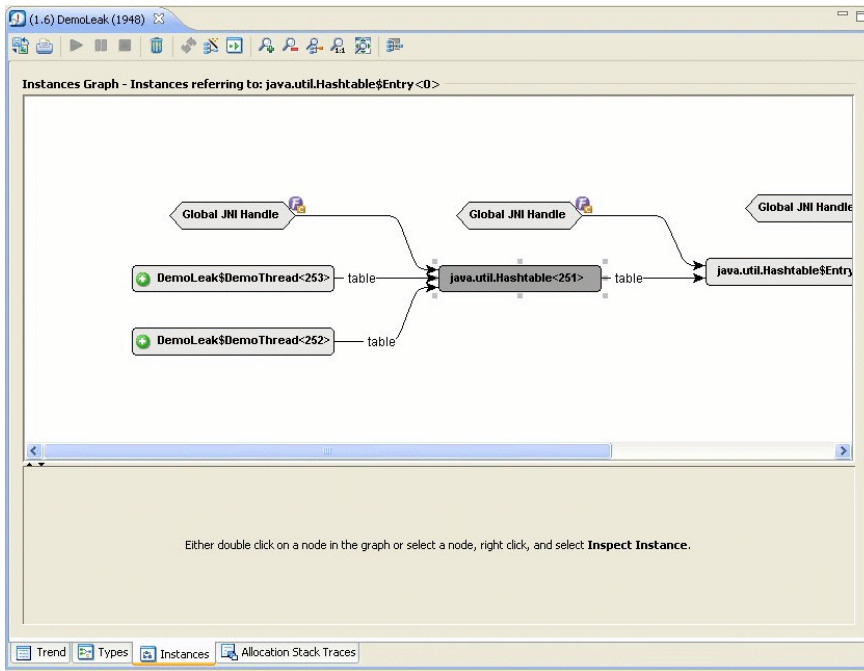
Figure 2-32 Show Referring Instances



At the bottom of the Types tab Fiona gets a list of instances of `java.util.Hashtable$Entry` that refer to `DemoObjects`.

8. Fiona right-clicks the topmost instance and selects “Show Referring Instances” (Figure 2-32) to start analyzing the instances that hold on to this particular hashtable entry.

Figure 2-33 Instances Tab



This opens the Instances tab, as seen in Figure 2-33.

9. Just as in the Types tab, Fiona clicks on the plus signs to expand the graph to the left. Fiona finds that the `DemoObject` is stored in a hashtable, which is held by a `DemoThread`. This is the culprit causing the memory leak.

The Leak is Discovered

Judging from the evidence she collected using the Oracle JRockit Mission Control tools, Fiona was able to not only identify her system problem as a memory leak, but was able to locate exactly which object type was leaking the memory. The key to making the identification began with

noting in her JRA recording the increasing length of garbage collections and the types upon which those lengthening garbage collections were occurring. She then ran the Memory Leak Detector to pinpoint which of the questionable types were actually the source of the leak. She was able to spot types that continued to increase in their number of instances and allocations, obviously holding on to memory so that it couldn't be freed for allocating other objects. This, then identified the memory leak and where it was occurring.

Oracle JRockit Mission Control Client Use Cases