BEA JRockit Mission Control

JRockit Management Console User Guide (JRockit 1.4.2 R26)

1.4.2
August 2006

Copyright

Restricted Rights Legend

Trademarks and Service Marks

Contents

Introduction

Using the BEA JRockit Management Console

Adding Custom Notification Actions and Constraints

Tracing Thread Activity With Stack Dumps

Index

CHAPTER 1

Introduction

Welcome to Using BEA JRockit Management Console. This document contains procedures and other information necessary for you to gain optimal performance from BEA Systems' industry-leading Java Virtual Machine, BEA JRockit.

This Introduction includes information on the following subjects:

What's In the User Guide?

This user guide is organized as follows:

- Using the BEA JRockit Management Console shows how to monitor and control running instances of BEA JRockit JVM using this graphic tool. The management console provides real-time information about the running application's characteristics, which is helpful both during development and in a deployed environment
- Adding Custom Notification Actions and Constraints shows you how to create custom notification actions and constraints for the Management Console.
- Tracing Thread Activity With Stack Dumps shows how to extract thread activity information from stack traces.

Information About JRockit Compatibility

The JRockit Management Console, described in this document, is compatible with JRockit 1.4.2 R26.2 and later.

Note: To monitor a Java application running on JRockit 1.4.2, you must use the Management Console bundled with JRockit 1.4.2. Attaching a 5.0 console to a JRockit 1.4.2 process does not work.

Finding Additional Information

You can find additional information about BEA JRockit throughout the BEA JRockit documentation set. For a complete list of available documents, please refer to the BEA JRockit JDK Online Documentation.

CHAPTER 2

Using the BEA JRockit Management Console

The JRockit Management Console can be used to monitor and control running instances of BEA JRockit JVM. It provides real-time information about the running application's characteristics, which can be used both during development—for example, to find where in an application's life cycle it consumes more memory—and in a deployed environment—for example, to monitor the system health of a running application server.

This section includes information on the following subjects:

Console Overhead

The extra cost of running the JRockit Management Console against a running BEA JRockit JVM is very small and can almost be disregarded. This provides for a very low cost monitoring and profiling of your application.

Note: It is not recommended that you run the Management Console on the same machine as the VM you are monitoring. If you run the Console on the same machine as the BEA JRockit you are monitoring, the Management Console GUI will steal valuable resources from the application running on the JVM and you risk performance degradation as a result.

Starting the Console

Starting the Management Console is a two-step process:

1. Enable the Management Server
2. Start the JRockit Management Console

Additionally, you might want to also complete these tasks as part of the start-up process:

- Set the Port
- Change the Number of Connections

Enable the Management Server

Before the Management Console can connect to BEA JRockit JVM, the management server in the VM needs to be started. The server is disabled by default. To enable the management server, start BEA JRockit JVM with the `-Xmanagement` option:

```
-Xmanagement
```

Attaching a Management Client

You can use the class= and `classpath=` parameters with `-Xmanagement` to specify a management class and its classpath; for example:

```
-Xmanagement:class=<classname>,classpath=<path>
```

This option loads the class and causes its empty constructor to be called early in JVM startup. From the constructor, a new thread is then started, from which your management client is run. You should ensure that the constructor returns control quickly because this call is made early in BEA JRockit startup.

Start the JRockit Management Console

Start the JRockit Management Console from the command prompt by typing:

```
console
```

Note: Before starting the Management Console, you must specify the JRE path and the classpath to the `.jar` file.

You can also start the Management Console without using the launcher. At the command line, enter:

`java -jar <jrockit-install-directory>/console/ManagementConsole.jar`

Starting the Management Server with a Security Manager

If you try to start the management server (`-Xmanagement` option) with a security manager running (`-Djava.security.manager` option) the management server might not start and you will get error messages such as the following:

```
"ERROR: failed to initialize class com.jrockit.management.rmp.
   RmpSocketListener."
```

To allow the management server to run under a security manager, add the text shown in Listing 2-1 to your policy file. The standard location of the policy file is:

- `java.home/lib/security/java.policy` (Linux)
- `java.home\lib\security\java.policy` (Windows)

For more information on policy files please refer to:

`http://java.sun.com/products/jdk/1.2/docs/guide/security/PolicyFiles.html`

Listing 2-1 Code for Starting the Management Server with a Security Manager

```
/* --- Permissions for the JRockit management Server --- */


/* TODO 1: Locate the installed managementserver.jar in JAVA_HOME/jre/lib */
grant codeBase "file:C:/MY_JAVA_HOME/jre/lib/managementserver.jar" {

   /* TODO 2: Add permissions for your console client to connect.  */
   permission java.net.SocketPermission "my-console-client.com", "accept,
   resolve";

   /* TODO 3: Add permissions for the management server to listen for
      connections.  */
   permission java.net.SocketPermission "localhost:7090", "listen,
      resolve";
```

```
  /* Add permissions for management server standard operations.  */
  permission com.bea.jvm.ManagementPermission "createInstance";
  permission java.lang.RuntimePermission "modifyThreadGroup";
  permission java.lang.RuntimePermission "modifyThread";
  permission java.lang.RuntimePermission "shutdownHooks";
  permission java.util.PropertyPermission "*", "read, write";
};
```

Set the Port

When BEA JRockit JVM is started with the `-Xmanagement` option set—and provided the VM is not running in "quiet" mode—it should print out a short message following the command line indicating that the management server is running and which port it is using. You can optionally choose which port to use by setting, as a command line argument, the port number in the `port` property:

```
java -Djrockit.managementserver.port=<portnumber>
```

The default port the management server uses to connect is 7090. It is strongly recommended that you block this port in your firewall, otherwise unauthorized users might access the management server.

Change the Number of Connections

You can change the number of connections allowed to the server by setting the `maxconnect` property:

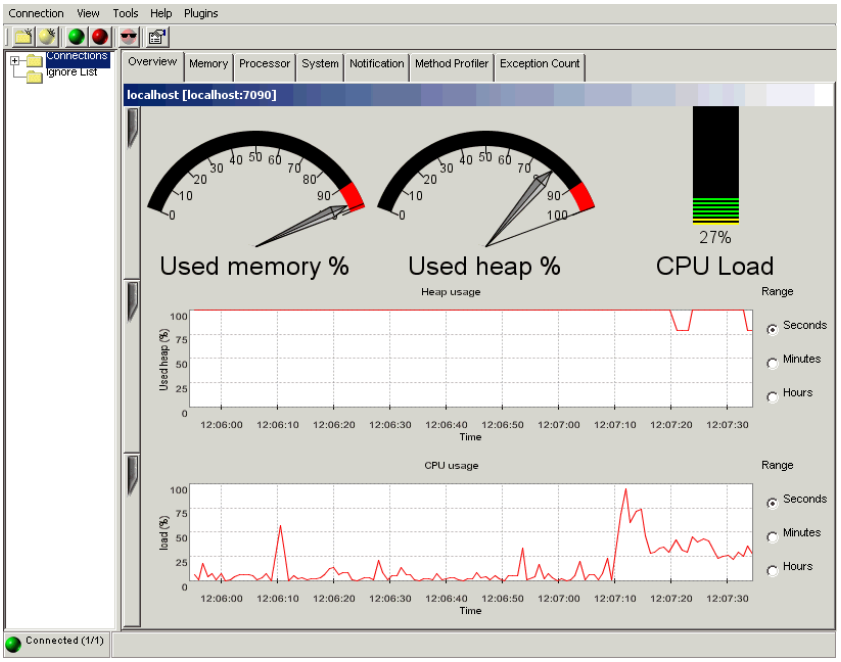
```
-Djrockit.managementserver.maxconnect=<maximum number of connections>
```

The default limit is four concurrent connections. While this should be enough for most users, you can change it, if necessary. The connection limit protects against Denial of Service (DoS) attacks by intruders.

Parts of the Console

When the JRockit Management Console window appears, the console has started, as shown in Figure 2-1:

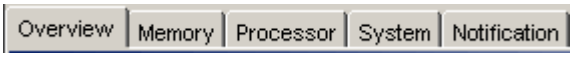
Figure 2-1 BEA JRockit JVM Management Console



The JRockit Management Console window is divided into two panes: a connection browser tree in the left pane (Figure 2-2) and a tabbed interface in the right pane (Figure 2-3).

Figure 2-2 Connection Browser

Figure 2-3 Information Tabs (Administrator Mode)



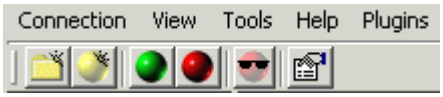
The first tab shows an Overview of information for the selected BEA JRockit JVM connection(s) (as highlighted in the connection browser pane). The other tabs contain detailed information about different areas of the VM, as will be described in Information Tabs.

Figure 2-3 shows the information tabs available in the console's Administrator operation mode. When the console is in the Developer mode, additional tabs appear, as shown in Figure 2-4. These two operation modes are described in Setting the Operation Mode.

Figure 2-4 Information Tabs (Developer Mode)

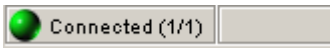
The console includes a toolbar that contains command buttons for some of the menu options (Figure 2-5). To toggle the Toolbar on or off, on the View menu select Tool Bar.

Figure 2-5 Management Console Toolbar



The status bar (Figure 2-6) at the bottom of the window displays informational messages and tool tips when you hover over a toolbar button or select something in a menu. It also indicates whether the JRockit Management Console is connected to one or several BEA JRockit JVM implementations or not. To toggle the Status Bar on or off, on the View menu, select Status Bar.

Figure 2-6 Status Bar



Setting Up the Console

Once the console is running, you will need to configure it to suit your needs. Configuring—or "setting up"—the console includes these tasks:

- Creating Connections
- Enabling Console Settings

Creating Connections

The connection browser displays a collection of saved connections to BEA JRockit JVM organized in folders. If necessary, you can add your own folders and connection nodes to the tree structure. Active connections currently connected to a running VM are indicated by a green icon; those disconnected are indicated by a red icon.

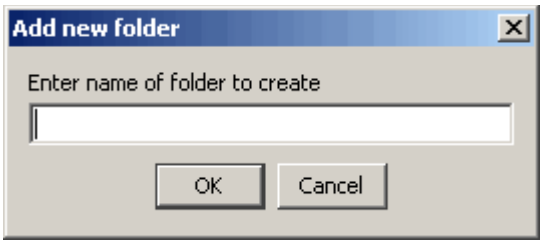
Creating a New Folder

To create your own folder in the connection browser, do the following:

1. Select an existing folder (for example, Connections) for which you want to create a subfolder.
2. Open the New Folder dialog box by doing one of the following:
   - Choose Connection→New Folder.
   - Press the right mouse button to open a context menu and select New Folder.
   - Press `Ctrl+N`.
   - Click the New Folder button on the toolbar.

   The Add new folder dialog box (Figure 2-7) appears:

Figure 2-7 Add New Folder Dialog Box



3. Enter the name of the new folder in the text field and click OK.

   The new folder will appear in the connection browser.

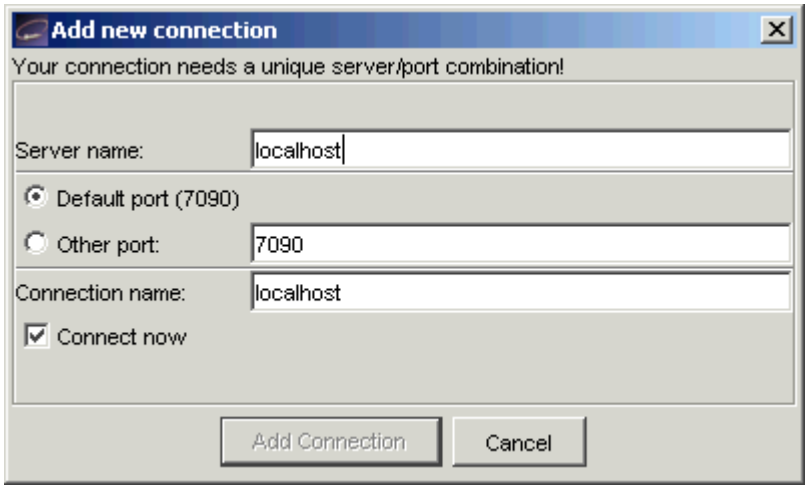
Creating a New Connection

To create a new connection to BEA JRockit JVM in the connection browser, do the following:

1. Select the folder in which the connection should be placed
2. Open the New Connection dialog box by doing one of the following:
   - Open the Connection menu and select New Connection.
   - Press the right mouse button to open a context menu and select New Connection.
   - Click the New Connection button on the toolbar.

   The Add new connection dialog box (Figure 2-8) appears:

Figure 2-8 Add New Connection Dialog Box



3. Enter the name of the server, the port and the new connection in the appropriate text fields or retain the default values. Then, select or deselect Connect now and click Add Connection.

Connecting a Connection to BEA JRockit JVM

To connect to BEA JRockit, do the following:

1. Select the BEA JRockit JVM connection to connect, a subfolder of connections to connect, or the folder Connections to connect all existing connections.
2. Do one of the following to connect the selected connection(s):
   - Open the Connection menu and select Connect.
   - Press the right mouse button to open a context menu and select Connect.
   - Press `Ctrl+O`.
   - Click the Connect button on the toolbar.

   When the connection is made, the status bar will read “Connected” and activity on the console will commence.

Disconnecting a Connection to the BEA JRockit JVM

To disconnect a connection to the BEA JRockit JVM, do the following:

1. Select the BEA JRockit JVM connection to connect, a subfolder of connections to connect, or the folder Connections to disconnect all existing connections.
2. Do one of the following to disconnect the selected connection(s):
   - Open the Connection menu and select Disconnect.
   - Press the right mouse button to open a context menu and select Disconnect.
   - Press `Ctrl+D`.
   - Click the Disconnect button on the toolbar.

   The connection will be lost and the status bar will indicate that you've been disconnected. All activity on the console will cease.

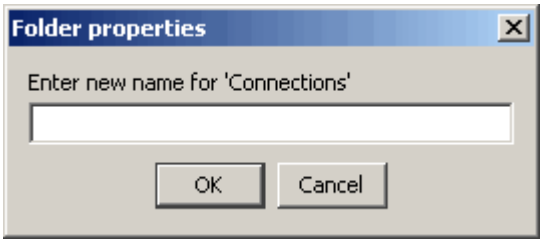
Renaming a Connection or Folder

To rename a connection or a connection folder , do the following:

1. Select the BEA JRockit JVM connection or folder to rename.
2. Do one of the following to rename the selected connection or folder:
   - Open the Connection menu and select Properties.
   - Press the right mouse button to open a context menu and select Properties.
   - Press `F2`.
   - Click the name label of the item (see **Note**, below).

   The Folder properties dialog box (Figure 2-9) appears:

Figure 2-9 Folder Properties Dialog Box



3. Enter a new name into the text field and click OK

Note: If you select the last option (click the item label), the Folder properties dialog box will not appear. Instead, the label itself will be enabled for direct editing. Simply type the new name over the old and click away from the label or press Enter.

Removing a Connection or Folder

To remove a connection or folder, do the following:

1. Select a connection or a subfolder to remove.
2. Do one of the following to remove the selected item:
   - Open the Connection menu and select Remove.
   - Press the right mouse button to open a context menu and select Remove.
   - Press `Delete`.
3. Click Yes on the confirmation dialog box that appears.

   The selected item disappears from the connection browser.

Hiding Disconnected Connections

Sometimes you might want to show just information about active BEA JRockit JVM connections. To hide information about disconnected connections, do one of the following:

- Open the View menu and select Hide Disconnected.
- Click the Hide Disconnected button on the toolbar.

To show the information about disconnected connections again, simply deselect Hide Disconnected in same way that you made the selection.

Enabling Console Settings

This section describes how to enable various JRockit Management Console settings.

Setting the Operation Mode

The Management Console can be run in two different operating modes:

- **Administrator Mode**; This is the default mode, designed for system administrators who are interested in observing the state of the BEA JRockit JVM.

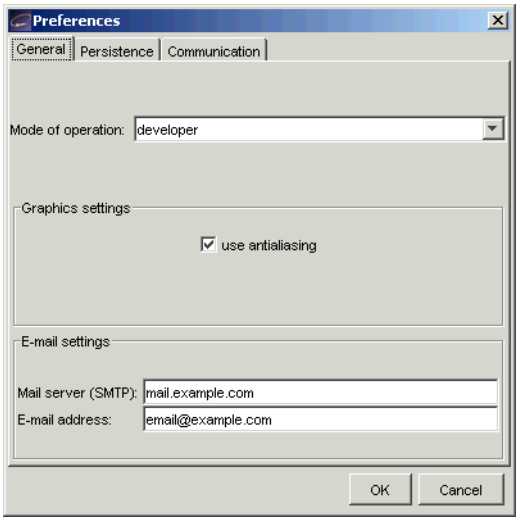
- **Developer Mode**; The developer mode is for developers and provides additional features such as a rudimentary method profiler and exception count functionality. Additional pages appearing in the developer mode are the Method Profiler page and the Exception Count page.

To set the operation mode, do the following:

1. From the Tools menu, select Preferences...

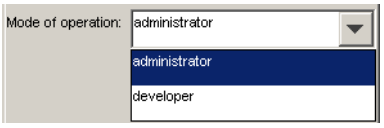
   The Preferences dialog box (Figure 2-10) appears:

Figure 2-10 Preferences Menu (General Tab)



2. Click the Mode of operation drop-down control to display the list of operation modes (Figure 2-11).

Figure 2-11 List of Operation Modes



3. Select the mode you want to use and click OK.

   Depending upon the mode to which you are toggling, the tabs on the console will change. See Figure 2-3 and Figure 2-4 for examples.

Setting Other Preferences

In addition to setting the operation mode, you can use the Preferences dialog box to change these settings:

- Default e-mail settings for the notification system (please refer to Notification Tab).
- Persistence behavior.

To change either of these values, open the Preferences dialog box from the Tools menu and proceed as described in the following sections:

Setting E-mail Preferences

To change e-mail preferences, do the following:

1. Select the General tab in the Preferences dialog box
2. In the appropriate text fields, enter the new e-mail information (SMTP server and E-mail address), as shown in Figure 2-12.
3. Click OK

Figure 2-12 E-mail Preferences Panel

Enabling Persistence

Enabling the persistence means that aspect values are saved to a file and can be reviewed in charts by opening the View menu and selecting View Historical Data (View Historical Data).

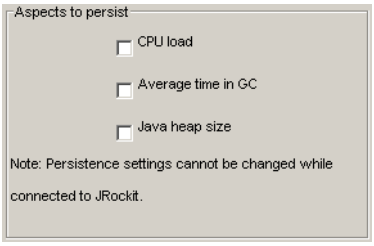
Selecting Aspects to Persist To set persistence preferences, do the following:

1. Disconnect any BEA JRockit JVM connections.

   Note: If you have not disconnected the connections and attempt to use this dialog box, you will be prompted to disconnect.

   The checkboxes in the Aspects to persist panel become enabled (Figure 2-13):

Figure 2-13 Aspects to Persist Panel



2. Select the aspects you want to persist.
3. Click OK.

   The selected aspect values are saved to a file that you can review in charts as described in "View Historical Data" on page 2-28.

Specifying the Persistence Directory In addition to setting preferences for the aspects to persist, you can also specify where to save the file that contains the aspect value (the "Persistence directory"). To do so:

1. Click Choose (next to the Persistence directory field).

   If you are still connected to BEA JRockit JVM, you will be prompted to disconnect; click Yes to proceed. A standard Open dialog box appears.
2. Locate the directory where you want to save the file and click Open.

   The Open dialog box closes, returning you to the Preferences dialog box.
3. Click OK.

   The new Persistence directory will appear in that field.

Erasing Persistence Value Logs Finally, you can erase all persistence value logs by clicking Clear all aspect logs. You will see a confirmation message to which you should respond Yes. Be aware that, if you delete all persistence value logs by clicking this button, you will also delete any other files stored in the `<USER_HOME>/console/data directory`.

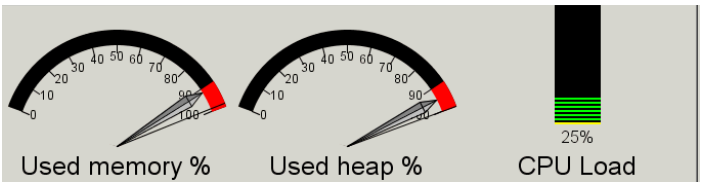
Customizing the Display

You can customize the console and change the way some of the monitoring data is displayed, as described in this section.

Customizing Gauges and Bars

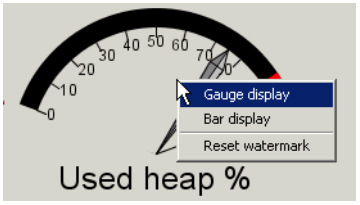
The gauges and bars are graphical devices showing memory and processor usage (Figure 2-14).

Figure 2-14 Gauges and Bars



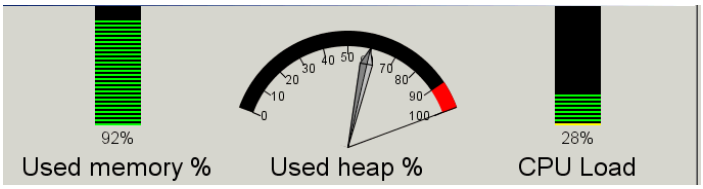
- To change from a gauge display to bar display, press the right mouse button when pointing at the gauge and select Bar display, as shown in Figure 2-15.

Figure 2-15 Gauge Context Menu (Bar Display Selected)



The selected gauge will appear as a bar (Figure 2-16).

Figure 2-16 Gauges and Bars with Gauge Converted to a Bar Display



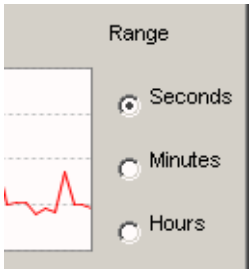
- To change back to a gauge, repeat the above, but select Gauge display.
- To reset the watermark—which indicates the highest level measured so far—press the right mouse button when pointing at the gauge or bar and select Reset Watermark.

Customizing Charts

Charts appear on the JRockit Management Console to show specified information about BEA JRockit.

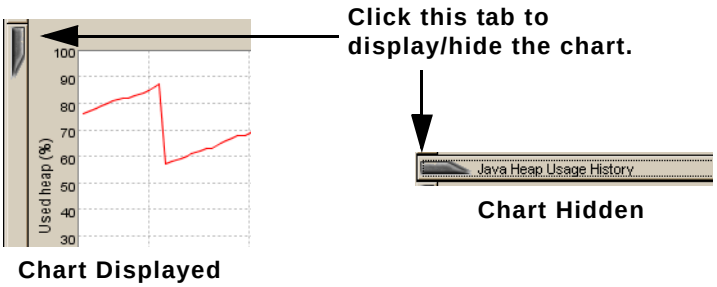
- To change scale on any of the charts, select the desired scale unit (seconds, minutes or hours) to the right of the chart (Figure 2-17) to be changed.

Figure 2-17 Range Selection Radio Buttons



- To hide a chart click the vertical tab to the left of the diagram you want to hide. When the diagram is hidden, the tab appears horizontally (Figure 2-18).

Figure 2-18 Hiding a Chart



- To show the diagram again, click the horizontal tab again.

Using the Settings File

When you exit the JRockit Management Console, your settings are automatically saved in a file called `consolesettings.xml`. This file is located in the folder:

```
<user home directory>\ManagementConsole
```

The exact path to the user home directory will vary on different platforms. On Windows it is usually something like `\Documents and Settings\<username>`; for example:

```
C:\Documents and Settings\jsmith\ManagementConsole
```

If no settings file exists in this directory it will be automatically created the next time the Management Console is closed.

WARNING: Do not edit this file by hand! Doing so can make it unusable and may cause the Management Console to crash on startup.

If you are experiencing problems with the settings file, you can always delete it and let the Management Console create a new one for you.

Using the Console

The JRockit Management Console monitors different “aspects” of the BEA JRockit JVM. An aspect is data that can be measured; for example, used heap size or VM uptime.

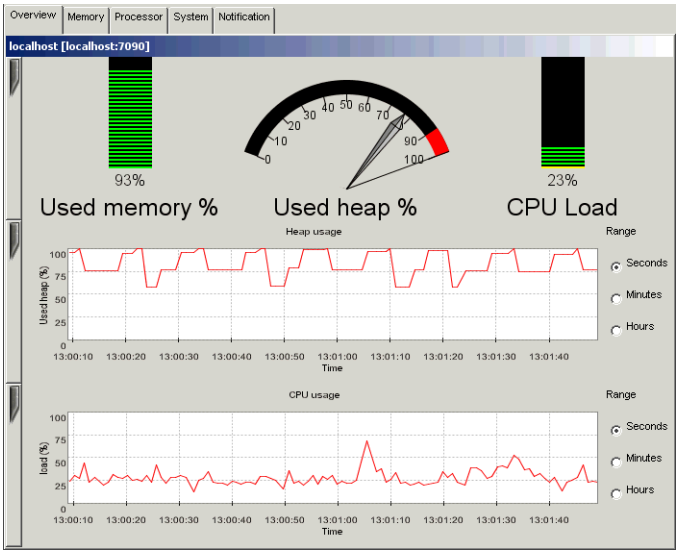
Information Tabs

Information tabs are pages containing details about different areas of the monitored BEA JRockit JVM. Display a tab by clicking it or by accessing the View menu. This section describes the tabs available on the JRockit Management Console.

Overview Tab

The Overview tab (Figure 2-19) shows an overview of selected connections. To select more than one connection, select the folder containing the connections you want to view. They will appear simultaneously. The page is divided into a “dashboard” with gauges in the upper part and charts in the lower part.

Figure 2-19 Overview Tab

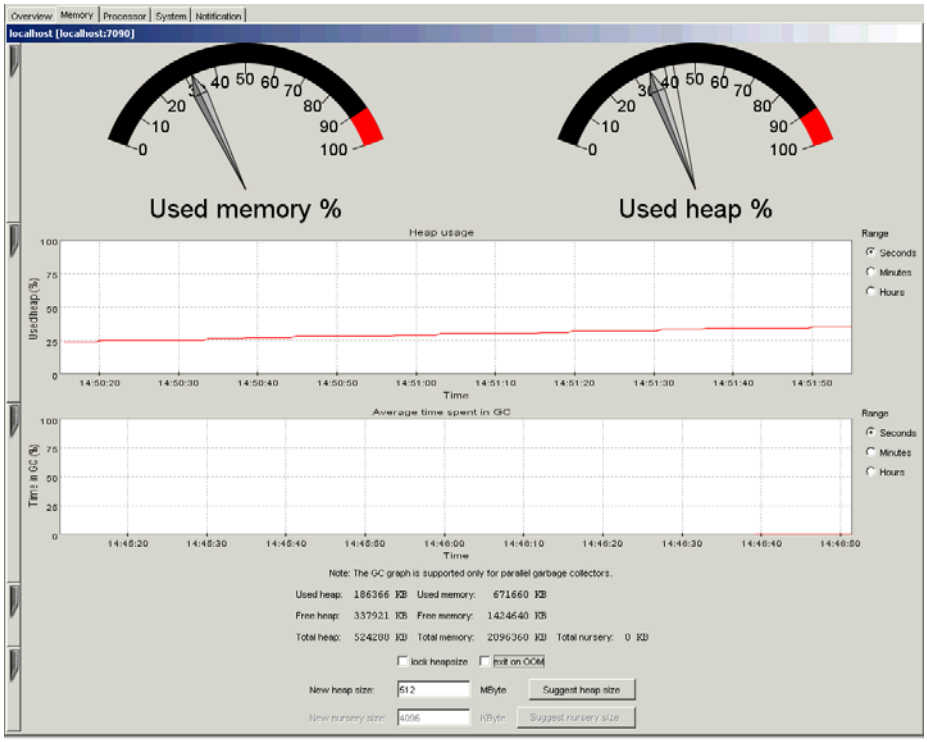


- The **Used Memory** gauge shows the percentage of occupied physical memory on the computer.
- The **Used Heap** gauge shows the percentage of occupied Java heap memory in the VM.
- The **CPU Load** bar shows the usage rate of the processor - or the average processor load on a multi-processor machine.
- The **Heap Usage** chart shows the percentage of used Java heap over time.
- The **CPU Usage** chart shows the average usage rate of the processor(s) over time.

Memory Tab

The Memory tab (Figure 2-20) shows information about the memory status of the system, as shown.

Figure 2-20 Memory Tab



- The **Used Memory** gauge shows the percentage of machine memory in use.
- The **Used Heap** gauge shows the percentage of occupied Java heap.
- The **Heap Usage** chart shows the percentage of occupied heap over time.
- The **Time in GC** chart shows the average time spent on garbage collection over time. This chart is only updated when running the BEA JRockit JVM with the Parallel garbage collector, and an actual garbage collection occurs.

At the bottom of the page the following text information is displayed (in kilobytes):

- **Used Heap** shows the occupied heap space.
- **Free Heap** shows the free heap space.
- **Total Heap** shows the heap size.

- **Used Memory** shows the amount of occupied physical memory.
- **Free Memory** shows the amount of free physical memory.
- **Total Memory** shows the total physical memory size.

Memory Tab Functionality

You can manipulate certain memory aspects of the JVM from the Memory Tab. These aspects are described in Table 2-1

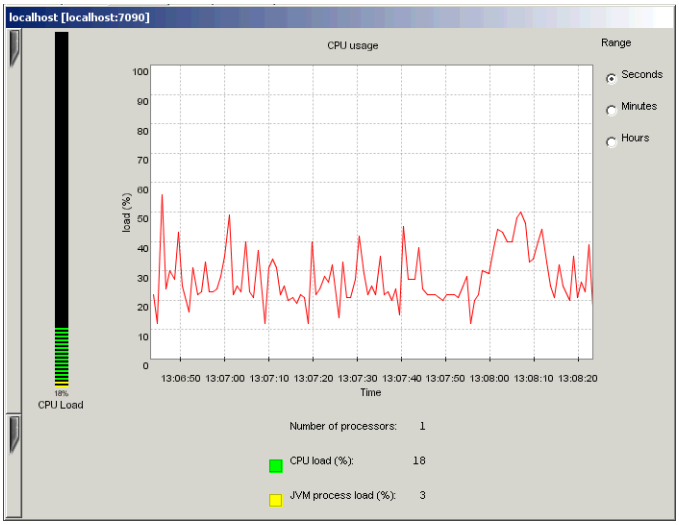
Table 2-1 Memory Tab Functionality

Function	Procedure
Manipulating the Heap Size	You can manipulate the heap size in any of the following ways: • To reset the size of your heap, enter a numeric value in the **New heap size** field and click **Suggest heap size**. The size set here, expressed in megabytes, represents the current heap size, not the maximum heap size. • To make the current size the maximum heap size, click **lock heap size**. Please refer to Setting the Heap Size for heap size requirements.
Changing the Nursery Size	If you are running a generational garbage collector, you can reset the nursery size by typing a new value in **New nursery size** and click **Suggest nursery size**. **Note:** If you are not running a generational collector, these fields will appear disabled. Please refer to Setting the Size of the Nursery for nursery size requirements.
Exiting on Out of Memory Errors	If you want to exit the JVM when you encounter an Out of memory (OOM) error, select **Exit on OOM**.

Processor Tab

The Processor tab (Figure 2-21) shows information about the processor status of the system.

Figure 2-21 Processor Tab



- The **CPU Load** bar shows the average processor load as a percentage. The overall load is displayed in green while the load of the JVM process(es) is displayed in yellow.
- The **CPU Usage** chart shows the average processor load as a percentage over time.

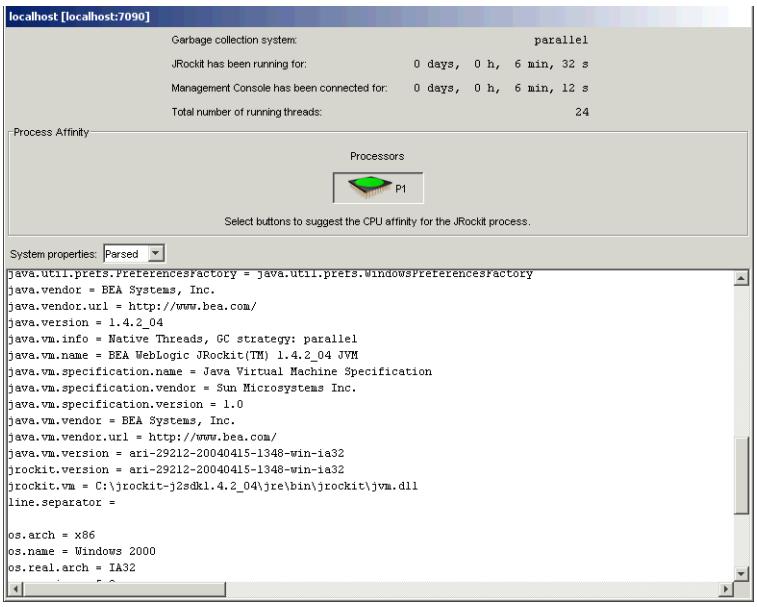
At the bottom of the page the following text information is displayed:

- **Number of Processors** shows the number of processors.
- **CPU Load** shows the overall processor load as a percentage.
- **JVM Process Load** shows the load of the BEA JRockit JVM process(es), expressed as a percentage.

System Tab

The System tab (Figure 2-22) shows various information about the system status.

Figure 2-22 System Tab



- **Garbage Collection System** shows which garbage collector the BEA JRockit JVM is running. If you are using a dynamic garbage collector (`-Xgcprio`), this value will change when the garbage collector changes. For more information on the dynamic garbage collector, please refer to The Dynamic Garbage Collector.
- **JRockit has been running for** shows how long the BEA JRockit JVM has been running.
- **Management Console has been connected for** shows how long the currently displayed connection has been connected.
- **Total number of running threads** shows the number of active threads at any given time in the application run.
- **Process Affinity** contains buttons that correspond to processors. It displays a green icon if the BEA JRockit JVM is running on this processor and a red icon if it is not. By selecting a button, the BEA JRockit JVM process can be bound to one or more processors. The VM might be released from such a connection by deselecting the button again. This is only a suggested affinity: the operating system might not follow the suggestion. Changing the process affinity is a feature that is only available when monitoring a VM instance running on the Windows platform. The Process Affinity display is only activated when the Management Console is in the Developer mode, described in Setting the Operation Mode.

- **System Properties** shows the Java System Properties loaded in the VM.

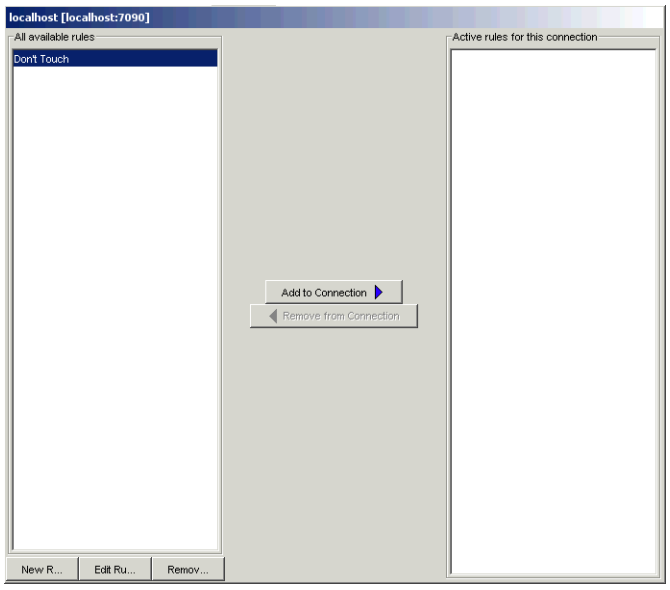
Notification Tab

Use the Notification tab (Figure 2-23) to define alerts that notify users when certain events occur. You can create your own notification rules based on different triggers, with optional constraints, that alert you with a prescribed notification. This section describes how to create these rules.

Creating Custom Actions and Constraints

After starting the Management Console for the first time, a file named `consolesettings.xml` will be created in the `\ManagementConsole` directory in your `<user_home>` directory. Among the contents of this file are the entries for the default actions and constraints. You can programatically create custom notification actions and constraints, which are also stored in this file. Once added, these actions and constraints will appear on the Notifications tab of the Management Console. For complete information on creating custom notification actions and constraints, see "Adding Custom Notification Actions and Constraints."

Figure 2-23 Notification Tab (No Rules Defined)



A notification trigger can be a certain event, for example, that the connection to the BEA JRockit JVM was lost, or that an aspect reaches a certain value, for example, the used memory reaches

95%. A notification constraint can limit when a rule is triggered for example by not sending alerts at night or on certain dates.

The notification action is how the alert is communicated to the user. It can be one of the following:

- E-mail shows an e-mail when the notification is sent to the specified address using the specified SMTP server.
- System out action displays the notification in the command window where you started the JRockit Management Console.
- Application alert displays the notification in an alert dialog in the Management Console.
- Log to file logs the notification to the specified file.

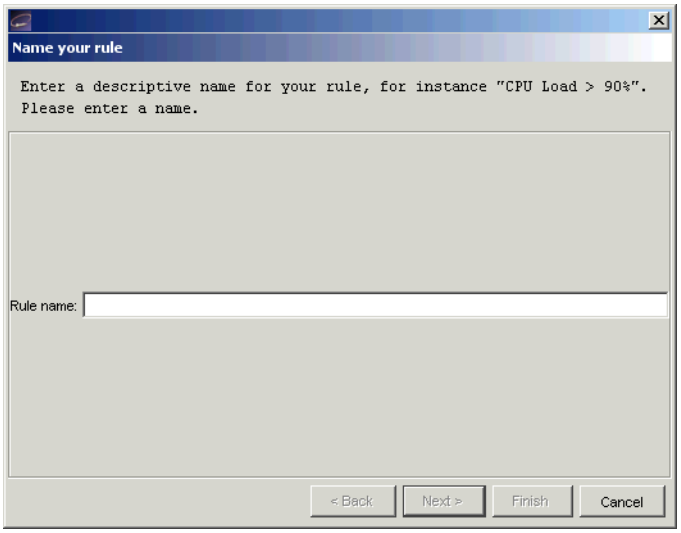
Creating a New Rule

Rules determine when and how to issue a notification. To create a new rule, do the following:

1. Click New Rule.

   The Name your rule dialog box appears (Figure 2-24):

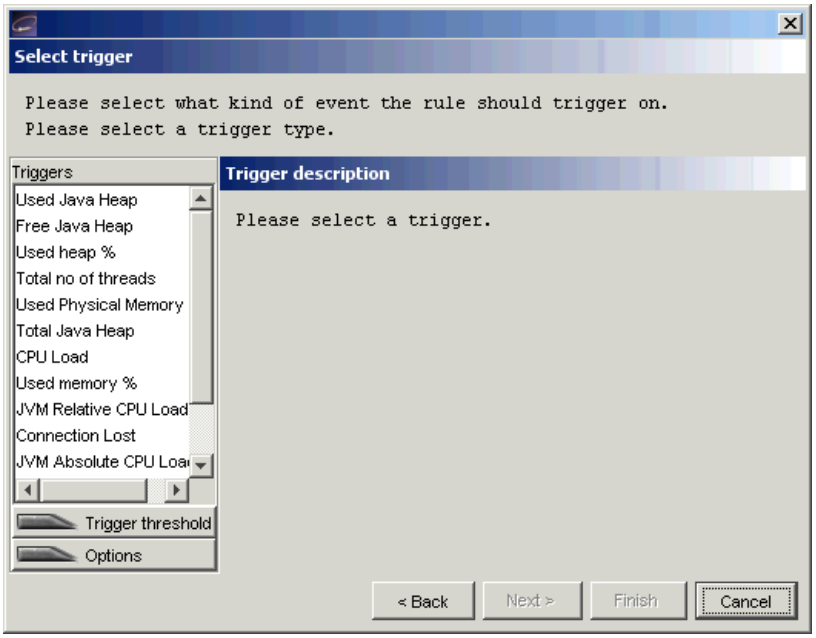
Figure 2-24 Name Your Rule Dialog Box



2. Enter the name of the new rule in Rule name: and click Next.

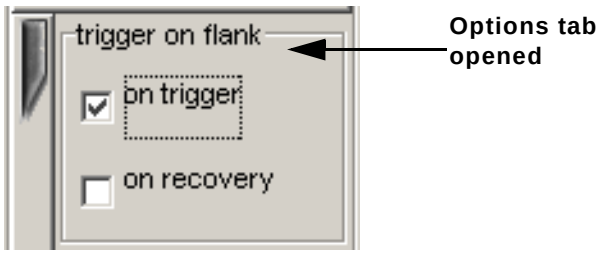
The Select trigger dialog box appears (Figure 2-25):

Figure 2-25 Select Trigger Dialog Box



3. Select a trigger (the individual triggers are described in the right panel).
4. Enter a threshold in the text box below the trigger list, if required (Figure 2-26; this box will be marked either Min value or Max value, depending on the type of trigger selected.

Figure 2-26Trigger Threshold and Options Text Boxes



5. Select further options under the Option tab. For example, in Figure 2-26, you need to select what kind of aspect value change will trigger the notification:

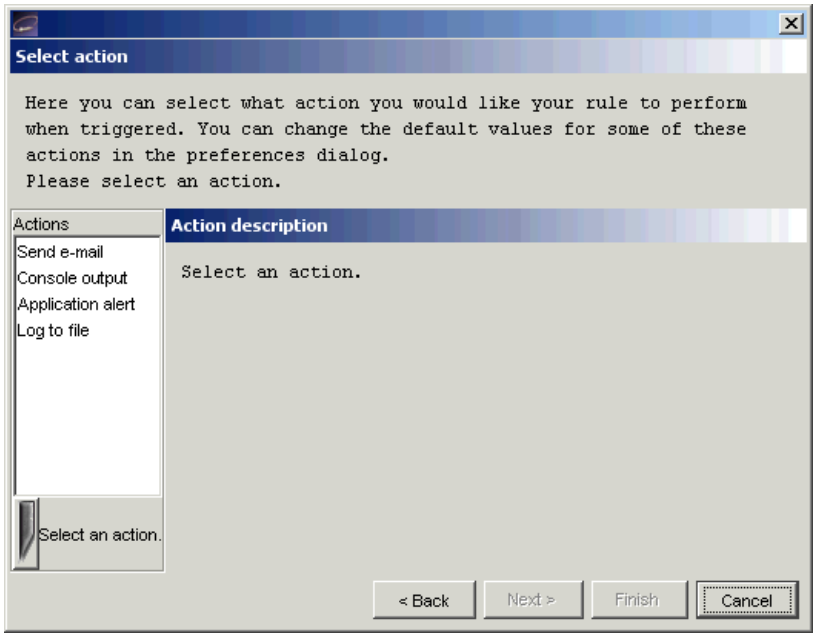
– on trigger, which triggers the notification when the aspect reaches the trigger value from a lower value (for example, if the trigger is 80 and the aspect value moves up from 75).

– on recovery, which triggers the notification when the aspect reaches the trigger value from a higher value (for example, if the trigger is 80 and the aspect value moves down from 85).

6. Click Next.

   The Select action dialog box appears (Figure 2-27):

Figure 2-27 Select action Dialog Box



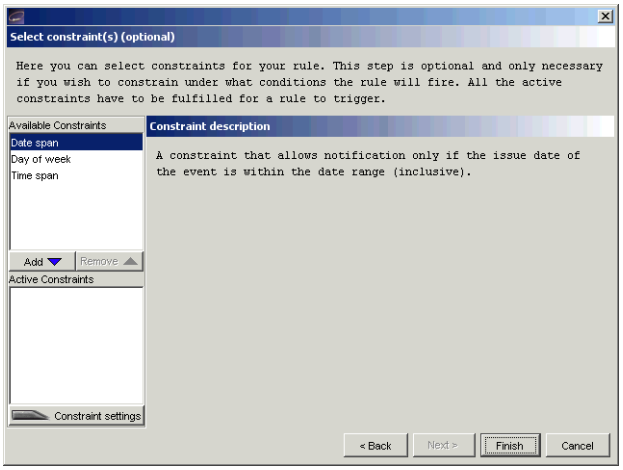
7. Select an action and enter settings data, if required.

8. If necessary, add a constraint to the rule (this step is optional; if you don't want to add a constraint, go to step 8):

   a. Click Next.

      The Select constraint(s) dialog box appears (Figure 2-28):

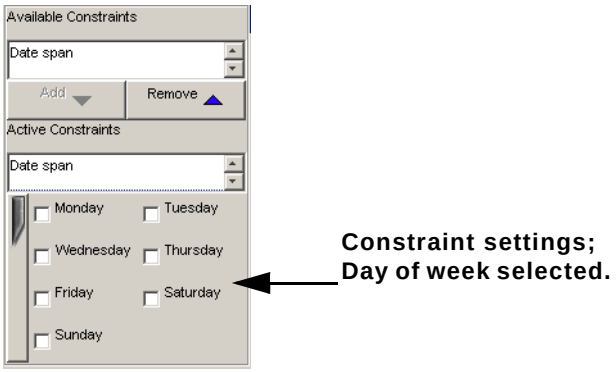
Figure 2-28 Select constraint(s) Dialog Box



b. Select a constraint and click Add.

The constraint name will appear in the add list, as shown in Figure 2-29.

Figure 2-29 Constraint Added

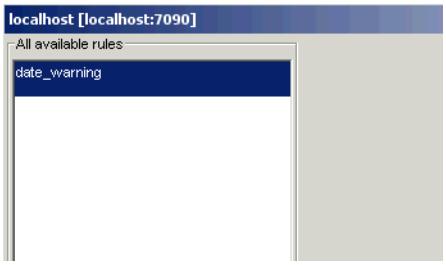


c. Enter constraint settings in the text fields under the list of constraints (Figure 2-29).

9. Click Finish.

The new rule appears in the All available rules list on the Notification tab, as shown in Figure 2-30.

Figure 2-30 New Rule in List



10. Add the rule to your connection as described in Add a Rule to BEA JRockit JVM.

Editing a Rule

To edit a rule, do the following:

1. In the Available rules list, select the rule to be edited and click Edit Rule.
2. Check the name of the rule, edit if necessary, and click Next.
3. Check the trigger and trigger settings, edit if necessary, and click Next.
4. Check the action and the action settings and edit if necessary.
5. To continue editing the rule, do the following (optional; if you don't want to add a constraint, go to step 6):
   a. Click Next.
   b. Check the constraints and the constraint settings and edit if necessary.
6. Click Finish.

Add a Rule to BEA JRockit JVM

To add a rule to the BEA JRockit JVM, do the following:

1. Select the rule to be added in the Available rules list.
2. Click Add to JRockit.

   The rule appears in the Active rules for this connection list, as shown in Figure 2-31.

Figure 2-31 Rule Added to Active rules for This Connection List

Remove a Rule from the BEA JRockit JVM

To remove a rule from the BEA JRockit JVM, do the following:

1. Select the rule to be removed in the Active rules for this connection list.
2. Click Remove from JRockit.

   The rule will now be removed from the Active rules for this connection list.

Remove a Rule

To remove a rule from the Available rules list, do the following:

1. Select the rule to be removed.
2. Click Remove Rule.

   A removal confirmation dialog box appears.
3. Click Yes
4. The rule disappears from the Available rules list.

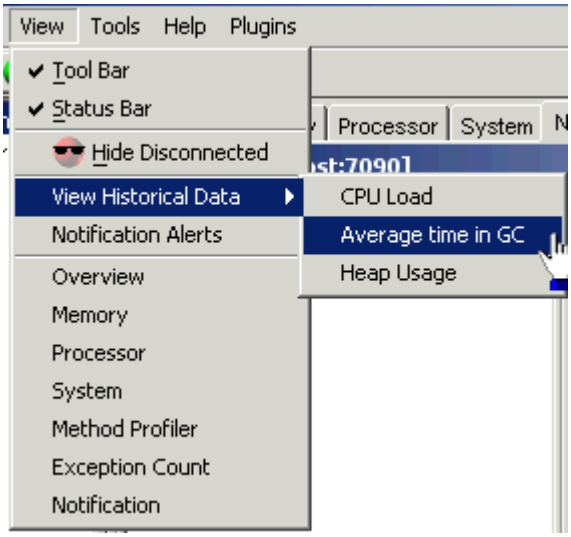
View Historical Data

The historical data window displays a chart where historical data for an aspect can be viewed. This is useful for observing trends over time and, for example, finding when a server running with a BEA JRockit JVM has its peak loads.

To open this window, do the following:

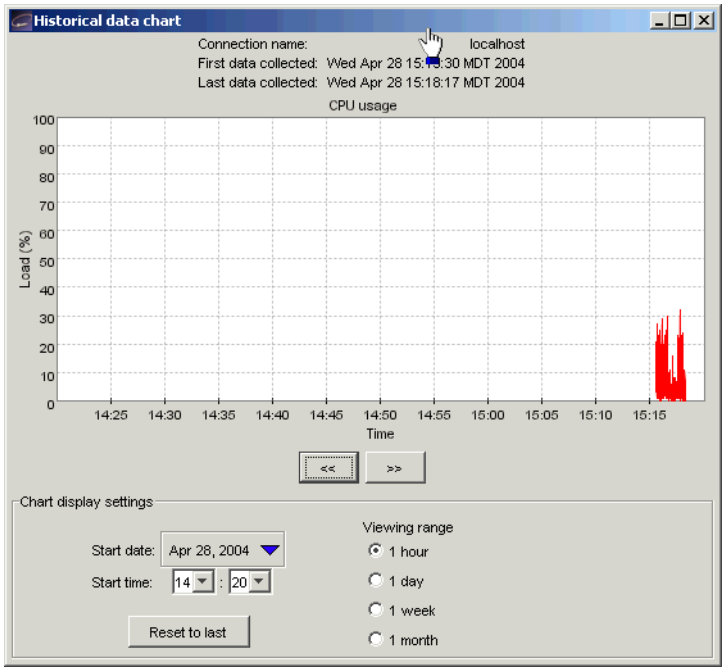
1. Select the connection for which you want to view data.
2. Open the View menu and select View Historical Data.
3. Select the aspect for which you want to view historical data, as shown in Figure 2-32.

Figure 2-32 View Menu with Historical Data Submenu Open



Historical data for the selected aspect appears (Figure 2-33).

Figure 2-33 Historical Data (CPU Load Selected)



4. Navigate through time either by using the arrows or changing the start time in the Chart display settings.

To observe historical data, aspect data from a BEA JRockit JVM must first have been persisted, that is, written to file. See Setting Other Preferences to enable or disable persistence. The following aspects are possible to persist, and thus display, historical data for:

- Used heap (as a percentage)
- CPU load (as a percentage)
- Average time spent garbage collecting (as a percentage)

As soon as data has been created by a connected connection, it is available for historical observation.

Using Advanced Features of the Console

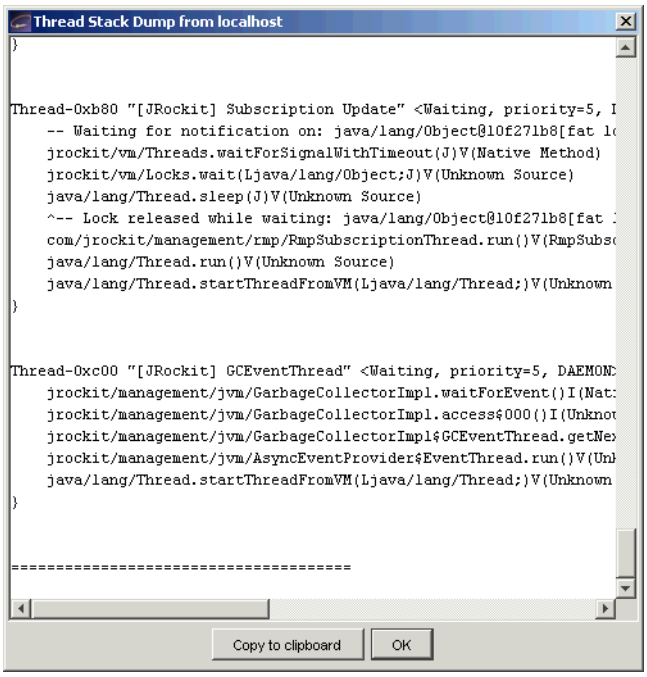
This section describes the more advanced features of the Management Console. Some of these are only available when running in the Developer mode, described in Setting the Operation Mode.

View Thread Stack Dump

The stack dump contains a list of all running threads in a BEA JRockit JVM with a method call stack trace for each thread.

To view the thread stack dump, open the Tool menu and select View Thread Stack Dump. A dialog box containing the stack dump appears (Figure 2-34).

Figure 2-34 Thread Stack Dump



Method Profiling Tab

Note: You must be in the developer operation mode before you can perform the tasks described in this section. For more information on entering the developer operation mode, see Setting the Operation Mode.

The Method Profiler tab allows the developer to monitor method execution in a non-intrusive way. The Method Profiler can provide information about the average time spent in selected methods and the number of times methods are invoked.

Method Templates are collections of methods that can be re-used on different connections. There is a Default template, but the user may also create new templates.

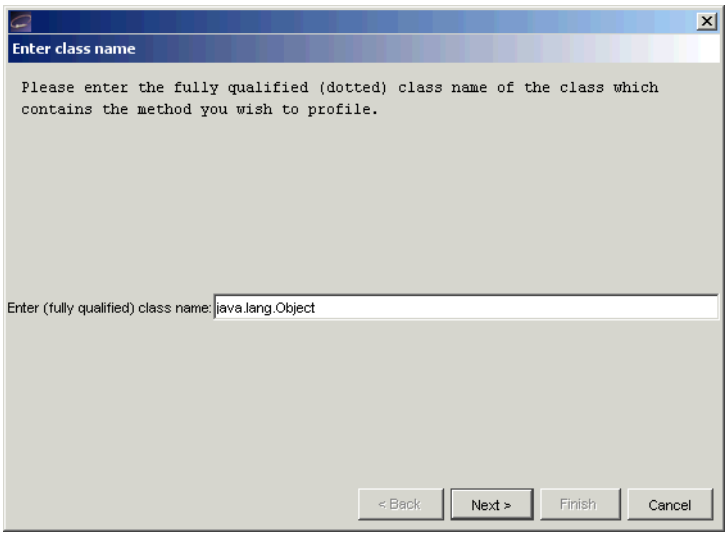
Adding a Method to a Template

To add a method to a template, do the following:

1. Select the template to be modified from the Select template list.
2. Click Add Method.

   The Enter class name dialog box appears (Figure 2-35).

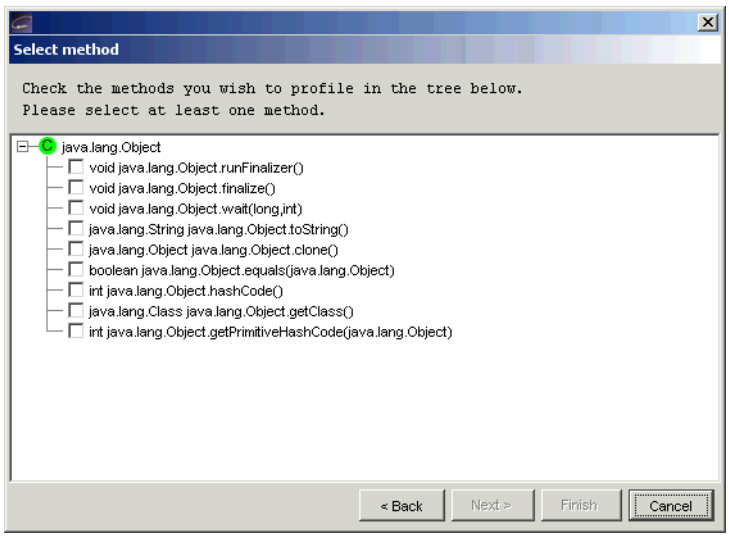
Figure 2-35 Enter Class Name Dialog Box



3. Enter a fully qualified class name, for example, `java.util.Vector`, in the text field and click Next.

   The Select method dialog box appears (Figure 2-36):

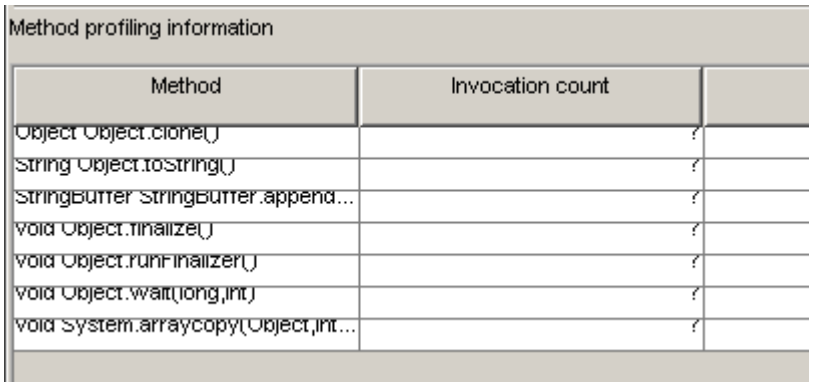
Figure 2-36 Select Method Dialog Box



4. Select the methods to be added to the template and press Finish.

   The method name will appear on the Method profiling information list, as shown in Figure 2-37.

Figure 2-37 Method Profiling Information List with Method Added



Removing a Method from a Template

To remove a method from a template, do the following:

1. From the Select template list, select the template you want to modify.

2. From the Method Profiling Information list, select the method(s) to be removed from the template.
3. Click Remove Method.

Creating a New Template

To create a new template, do the following:

1. Click New template.

   The New template dialog box appears (Figure 2-38).

Figure 2-38 New Template Dialog Box

2. Enter a name for the new template in the text field.
3. Click OK.

Removing a Template

To remove a template, do the following:

1. From the Select template list select the template to be removed.
2. Click Remove.

   A confirmation dialog box appears.
3. Click Yes.

Starting and Stopping Method Profiling

To start the method profiling, do the following:

1. From the Select template list, select the template to be started.
2. Click Start/Stop.

If you select Start, numbers in the Invocation count cells for each method begin to increment as method calls are made. If you select Stop, this activity will cease.

Method Profiling Settings

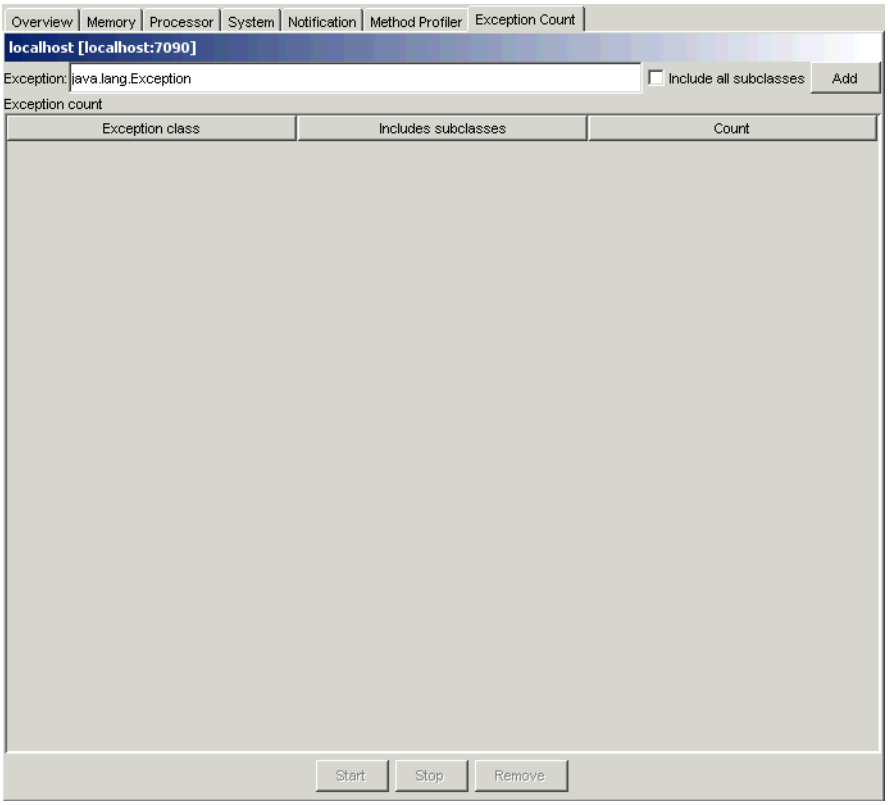
You can switch between using qualified method names or short names in the method profiling table.

- To enable invocation count, select the Invocation count checkbox at the bottom of the page.
- To enable timing, select the Timing checkbox at the bottom of the page.

Exception Counting Tab

The Exception Count tab (Figure 2-39) shows exceptions thrown in the BEA JRockit JVM. It counts the number of exceptions of a certain type thrown.

Figure 2-39 Exception Counting Tab



Add an Exception

To add an exception to observe, do the following:

1. Enter the fully qualified name of the exception into the text field at the top of the page, e.g., "`java.io.IOException`".
2. Choose whether or not all subclasses of that exception should be included in the count by selecting or deselecting the Include subclasses checkbox.
3. Click Add. You can only add subclasses of `java.lang.Throwable` which are loaded in the BEA JRockit JVM and you can only add exceptions while connected.

The exception should now be displayed in the table.

Starting, Stopping, and Removing an Exception Count

To start the exception count, click Start. The results should now appear next to the name of the exception being counted. Similarly, to stop the exception count, click Stop.

To remove an exception from the count, select the exception to be removed and click Remove.

Creating a JRA Recording

The BEA JRockit Runtime Analyzer (JRA) is an internal tool used by the BEA JRockit development team to analyze runtime performance of BEA JRockit and Java applications running on it. This tool provides information on internals in BEA JRockit that are useful to the development team.

One part of the JRA runs inside the JVM, recording information about it and the Java application currently running. This tool is launched from the Management Console, as described in the following procedure. The recorded information is saved to a file which you can view in the analyzer tool, as described in Using the BEA JRockit Runtime Analyzer.

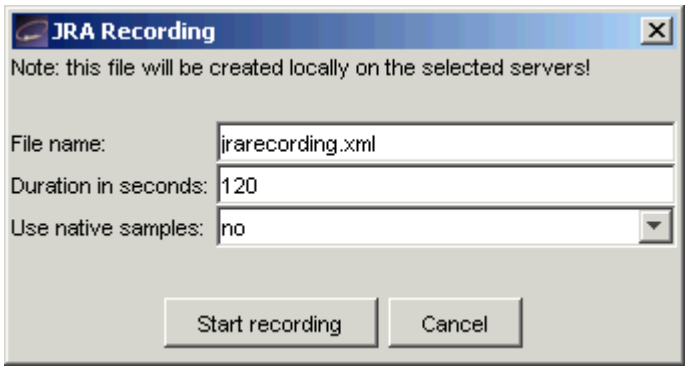
To make a recording, use this procedure:

Note: Before you can make a recording, you need to be working in the Developer mode, as described in "Setting the Operation Mode."

1. Open the Plugins menu and select Make a JRA recording.

   The JRA Recording dialog box appears (Figure 2-40).

Figure 2-40 JRA Recording Dialog Box



2. Type a descriptive name for the recording in the Filename field. This will be the name by which the file is saved.

Optionally, you can also select

- The duration of the recording (in seconds)
- Whether or not to use native samples.

3. Click Start recording.

   The JRA Recording Progress box appears (Figure 2-41).

Figure 2-41 JRA Recording Progress Box

4. When the recording is complete, click Done.

To view the recording, use the analyzer tool, as described in Using the BEA JRockit Runtime Analyzer.

Closing the Console

To close the JRockit Management Console and disconnect all connections, open the Connection menu and select Exit. Clicking X in the top right corner of the window will also close the JRockit Management Console.

Starting and Running the Console in the Headless Mode

You can run the Management Console, its notification subsystem, and the user actions without using a GUI. This function is referred to running the console in a "headless" mode and can greatly reduce the amount of system overhead required to run the Management Console.

Running a Headless Management Console

To run the Console in the headless mode, start the console as you normally would (see "Start the JRockit Management Console" for details) but add the -headless command-line option; for example:

```
java -jar ManagementConsole.jar -headless
```

You can control the console's behavior by using the command-line options described in Table 2-2.

As it runs, the JVM statistics normally associated with the Management Console can be written to file. The file to which statistics are written will be automatically created, but only if you choose to save, or "persist" data. It will be created in a directory of your choosing.

You can control which JVM statistics are persisted by specifying them in an XML settings file. The settings file is also created automatically, when you exit the application *when it is running in GUI mode*. By default, it will be created in the `<user_home>/.ManagementConsole` directory. You can specify another file at another location by using the `-settings` command-line option.

Controlling the Console with Command-line Options

You can use one of the command-line options listed in Table 2-2 to control the behavior of the headless Management Console.

Note: These options are not specific to running the Console in the headless mode; they are also valid when running it with a GUI.

Table 2-2 Headless Management Console Command-line Options

Option	Description
`-headless`	Starts the console in the headless mode (won't load GUI related classes).

Table 2-2 Headless Management Console Command-line Options

`-settings <settings file>`	Starts the console using the specified settings file. If you are starting in the GUI mode and this file doesn't exist, it will be created when you close the application.
`-connectall`	Connects to all connections in the setting file.
`-connect <connection 1> <connection 2> <...>`	Connects to the named connections available in the settings file previously added by using the GUI.
`-autoconnect`	Automatically connects to any JVM running the management server with JDP turned on.
`-uptime <time in seconds>`	Runs the Console for the specified amount of time, and then automatically shut it down.
`-jrockitmode`	Starts the Console in the JRockit Mode (only makes sense in GUI mode).
`-useraction <name> <delay in seconds> <period (optional)>`	Runs the named user action after the specified delay. If no period has been specified, the action will be run once. If the period has been specified it will be run every `<period (optional)>` seconds.
`-version`	Prints the version of the ManagementConsole and then exits.

For example:

```
java -jar ManagementConsole.jar -headless -settings
   C:\Headless\consolesettings.xml -connectall -autoconnect -uptime 3600
   -useraction ctrlbreak 30 60
```

This example

- Starts the management console in headless mode (`-headless`).
- Reads the specified settings file (`-settings C:\Headless\consolesettings.xml`).
- Tries to connect to all previously specified JVMs (`-connectall`).
- Actively searches for new connections using JDP (`-autoconnect`).

- After running 30 seconds, it will start issuing control breaks to all connected JVMs every minute (`-useraction ctrlbreak 30 60`).
- After an hour it will automatically shut down (`-uptime 3600`).

All notification rules that have been previously added to specific connections will be active.

APPENDIX A

Adding Custom Notification Actions and Constraints

After starting the BEA JRockit JVM Management Console for the first time, a file named `consolesettings.xml` will be created in the `\ManagementConsole` directory in your home directory. Among other entries, this file contains the deployment entries for the default actions and constraints. You can create custom notification actions and constraints for the Management Console, which are also stored in this file. Once added, these actions and constraints will appear on the Notifications tab of the Management Console.

This appendix includes information on the following subjects:

- Locating consolesettings.xml
- Creating a Custom Action
- Creating and Implementing an Action: Example
- Creating a Custom Constraint

Locating consolesettings.xml

The `consolesettings.xml` file is located in your home directory, under the `\ManagementConsole` folder. If you are using Windows, the path should be:

`C:\Documents and Settings\<user_name>\ManagementConsole`

(where *<user_name>* is the user name under which you are running the Management Console)

If you are using Linux, the path will normally be:

`/home/<user_name>/ManagementConsole`

(where <*user_name*> is the user name under which you are running Management Console)

Creating a Custom Action

The following procedure walks you through the steps necessary to create and implement a custom action. In this procedure, you will be creating a print action.

1. Add the `ManagementConsole.jar` to your build path.

   You can find this `.jar` in the `<jrockit_home>/console` directory.

2. Create a subclass of `AbstractNotificationAction`. This class will receive the `NotificationEvents`.

3. Implement `handleNotificationEvent`:

   ```
   public void handleNotificationEvent(NotificationEvent event)
   ```

   You can also override the `exportToXml` and `initializeFromXml` methods to store your action settings to XML.

4. Create a subclass of `AbstractNotificationActionEditor` to create the graphical editor used to edit the settings. If you have no editable settings for your action, you can just use the `com.jrockit.console.notification.ui.NotificationActionEditorEmpty`.

5. Implement the abstract methods:

   ```
   protected void storeToObject(Describable object);
   ```

   ```
   protected void initializeEditor(Describable object);
   ```

6. Edit the `consolesettings.xml` file to include your new action under the `<registry_entry>` element.

7. Add your new classes in the classpath.

8. Run the console.

The new action will be available in the new rule dialog box in the notification section of the Management Console (see Notification Tab).

Creating and Implementing an Action: Example

This section shows a real-life example of how an action is created and implemented. Once implemented, a text field where you can enter a parameter will appear on the Notification tab.

The step numbers that appear in headings below refer to the steps in the procedures under Creating a Custom Action.

Note: This example assumes that `ManagementConsole.jar` has been added to the build path.

Create the Action (Step 1)

First, we create a subclass of `AbstractNotificationAction`, as shown in Listing A-1. This class will receive the `NotificationEvents`.

Listing A-1 Building the Parameterized Action

```
package com.example.actions;
import org.w3c.dom.Element;

import com.jrockit.console.notification.*;
import com.jrockit.console.util.XmlToolkit;


/**
 * Test class showing how to build a parameterized action.
 *
 * @author Marcus Hirt
 */
public class MyTestAction extends AbstractNotificationAction
{
   private final static String TEST_SETTING = "test_param";
   public final static String DEFAULT_VALUE = "default value";
   private String m_parameter = DEFAULT_VALUE;


      /**
       * @see com.jrockit.console.notification.NotificationAction#
       * handleNotificationEvent(NotificationEvent)
       */
   public void handleNotificationEvent(NotificationEvent event)

   {
      System.out.println("===MyTestAction with param: " +
         getParameter() + "======");
```

```
      System.out.println(NotificationToolkit.prettyPrint(event));
   }
   /**
    * @see com.jrockit.console.util.XmlEnabled#exportToXml
    * (Element)
    */
   public void exportToXml(Element node)
   {
      XmlToolkit.setSetting(node, TEST_SETTING, m_parameter);
   }

      /**
       * @see com.jrockit.console.util.XmlEnabled#initializeFromXml
      * (Element)
      */
   public void initializeFromXml(Element node)
   {
      m_parameter = XmlToolkit.getSetting(node, TEST_SETTING,
         DEFAULT_VALUE);
   }
      /**
      * Returns the parameter.
      *
      * @return some parameter.
      */
   public String getParameter()
   {
      return m_parameter;
   }
   /**
   * Sets the parameter.
   *
   * @param parameter the value to set the parameter to.
   */
   public void setParameter(String parameter)
```

```
   {
      m_parameter = parameter;
   }
}
```

Implementing handleNotificationEvent() (Step 2)

While creating the subclass of `AbstractNotificationAction` created, we implemented handleNotificationEvent(), as shown in Listing A-2. This method acts on the incoming event.

Listing A-2 Implementing handleNotificationEvent

```
public class MyTestAction extends AbstractNotificationAction
{
   private final static String TEST_SETTING = "test_param";
   public final static String DEFAULT_VALUE = "default value";
   private String m_parameter = DEFAULT_VALUE;


      /**
       * @see com.jrockit.console.notification.NotificationAction#
       * handleNotificationEvent(NotificationEvent)
       */
   public void handleNotificationEvent(NotificationEvent event)
   {
```

Creating the Action Editor (Step 3)

Next, we create a subclass of `AbstractNotificationActionEditor` to create the graphical editor used to edit the settings. Listing A-3 shows how this is done.

Listing A-3 Creating the Action Editor

```
package com.example.actions;
```

```
import java.awt.*;
import javax.swing.*;

import com.jrockit.console.notification.Describable;
import com.jrockit.console.notification.ui.AbstractNotification
   ActionEditor;

/**
* Simple test editor. Displays a text field where you can enter a
* parameter.
* (Note that you'd get better layout results using a GridbagLayout.)
*
* @author Marcus Hirt
*/

public class MyTestActionEditor extends AbstractNotificationActionEditor

{
   private JTextField m_parameterField = new
   JTextField(MyTestAction.DEFAULT_VALUE);

   /**
   * Constructor for MyTestActionEditor.
   */
   public MyTestActionEditor()
   {
      super();
      setName("MyTestAction settings");
      add(new JLabel("Param:"), BorderLayout.WEST);
      add(m_parameterField, BorderLayout.CENTER);
      setMinimumSize(new Dimension(140,0));
   }
   /**
   * @see com.jrockit.console.notification.ui.Abstract
   * Editor#initializeEditor(com.jrockit.console.notification.
   * Describable)
   */
   protected void initializeEditor(Describable action)
   {
      m_parameterField.setText(((MyTestAction) action).
         getParameter());
```

```
   }
   /**
   * @see com.jrockit.console.notification.ui.AbstractEditor#
   * storeToObject(com.jrockit.console.notification.Describable)
   */
   protected void storeToObject(Describable action)
   {
      ((MyTestAction)action).setParameter(m_parameterField.
         getText());
   }
}
```

Implementing the Abstract Methods (Step 4)

When we created the action editor above, we implemented the abstract methods `initializeEditor()` and `storeToObject()`, as shown in Table A-4.

Listing A-4 Implementing the Abstract Methods

```
 */
   protected void initializeEditor(Describable action)
   {
      m_parameterField.setText(((MyTestAction) action).
         getParameter());
   }
   /**
   * @see com.jrockit.console.notification.ui.AbstractEditor#
   * storeToObject(com.jrockit.console.notification.Describable)
   */
   protected void storeToObject(Describable action)
   {
      ((MyTestAction)action).setParameter(m_parameterField.
         getText());
   }
```

Adding the New Action to the Deployment Entries (Step 5)

Before the action and editor can appear on the Management Console, you need to add it to the deployment entries in `consolesettings.xml`, under the `<registry_entry>` element, as shown in Listing A-5.

Listing A-5 Adding the New Action to the Deployment Entries

```
<registry_entry>
   <entry_class>
      com.company.actions.MyTestAction
   </entry_class>
   <entry_name>
      Test action
   </entry_name>
   <entry_description>
      Test action, dynamically added.
   </entry_description>
   <entry_editor_class>
      com.company.actions.MyTestActionEditor
   </entry_editor_class>
</registry_entry>
```

Displaying the New Action Editor (Steps 6 and 7)

Finally, add the new classes to your classpath and start the console. When you navigate to the Notifications tab, you'll see the new editor on the tab.

Creating a Custom Constraint

Create custom constraints by using the same procedure described in Creating a Custom Action, except that you must implement:

```
boolean validate(NotificationEvent event)
```

instead of:

```
void handleNotificationEvent(NotificationEvent event)
```

as shown in Listing A-6:

Listing A-6 Code Change for Creating a Customer Constraint

```
public class MyTestAction extends AbstractNotificationAction
{
   private final static String TEST_SETTING = "test_param";
   public final static String DEFAULT_VALUE = "default value";
   private String m_parameter = DEFAULT_VALUE;


      /**
       * @see com.jrockit.console.notification.NotificationAction#
       * handleNotificationEvent(NotificationEvent)
       */
      boolean validate(NotificationEvent event)
   {
```

APPENDIX B

Tracing Thread Activity With Stack Dumps

Stack dumps, or "stack traces," reveal information about an application's thread activity that can help you diagnose problems and better optimize application and JVM performance; for example, stack dumps can show the occurrence of "deadlock" conditions, which can seriously impact application performance.

Stack dumps usually occur when certain errors are thrown. You can also create a stack dump by invoking a control break (usually by pressing `Ctrl-Break` or `Ctrl-\`; or `SIGQUIT` on Linux). This section provides information on working with stack dumps. It includes information on these subjects:

- Monitoring Information in Stack Dumps
- Detecting Deadlocks

Monitoring Information in Stack Dumps

When printing stack traces with `Control-Break`, BEA JRockit also shows the status of active locks (monitors). For each thread, BEA JRockit prints the following information if the thread is in a waiting state:

If the thread is trying to take a lock (to enter a synchronized block), but the lock is already held by another thread, this is indicated at the top of the stack trace, as "Blocked trying to get lock".

If the thread is waiting on a notification on a lock (by calling `Object.wait()`), this is indicated at the top of the stack trace as "Waiting for notification".

If the thread has taken any locks, this is shown in the stack trace. After a line in the stack trace describing a function call is a list of the locks taken by the thread in that function. This is described as `^-- Holding lock` (where the `^--` serves as a reminder that the lock is taken in the function written above the line with the lock).

Caution: The lines with the lock information might not always be correct, due to compiler optimizations. This means two things:

- If a thread, in the same function, takes first lock A and then lock B, the order in which they are printed is unspecified.
- Sometimes, if a thread, in method `foo()` calls method `bar()`, and takes a lock A in `bar()`, the lock might be printed as being taken in `foo()`.

Normally, this shouldn't be a problem. The order of the lock lines should never move much from their correct position. Also, lock lines will never be missing—you can be assured that all locks taken by a thread are shown in the stack dump.

The semantics for waiting (for notification) on an object in Java is somewhat complex. First you must take the lock for the object, and then you call `wait()` on that object. In the wait method, the lock is released before the thread actually goes to sleep waiting for a notification. When it receives a notification, wait re-takes the lock before returning. So, if a thread has taken a lock, and is waiting (for notification) on that lock, the line in the stack trace that describes when the lock was taken is not shown as “Holding lock,” but as “Lock released while waiting.”

All locks are described as `Classname@0xLockID[LockType]`; for example:

```
java/lang/Object@0x105BDCC0[thin lock]
```

Where:

- `Classname@0xLockID` describe the object the to which the lock belongs. The classname is an exact description, the fully qualified class name of the object. `LockID`, on the other hand, is a temporary ID which is only valid for a single thread stack dump. That is, you can trust that if a thread A holds a lock `java/lang/Object@0x105BDCC0`, and a thread B is waiting for a lock `java/lang/Object@0x105BDCC0`, in a single thread stack dump, then it is the same lock. If you do any subsequent stack dumps however, `LockID` is not comparable and, even if a thread holds the same lock, it might have a different `LockID` and, conversely, the same `LockID` does not guarantee that it holds the same lock.
- `LockType` describes the kind of BEA JRockit internal lock type the lock is. Currently, three kinds of locks exist:

- fat locks: locks with a history of contention, or that have been waited on (for notification).
- thin locks: locks that have no contention (several threads trying to take the lock simultaneously).
- recursive locks: locks occur when a thread takes a lock it already holds.

Listing B-7 shows an example of what a stack trace for a single thread can look like.

Listing B-7 Example: Stack Trace for a Single Thread

```
"Open T1" prio=5 id=0x680 tid=0x128 waiting
   -- Waiting for notification on: java/lang/Object@0x1060FFC8[fat lock]
   at jrockit/vm/Threads.waitForSignalWithTimeout(Native Method)@0x411E39C0
   at jrockit/vm/Locks.wait(Locks.java:1563)@0x411E3BE5
   at java/lang/Thread.sleep(Thread.java:244)@0x41211045
   ^-- Lock released while waiting: java/lang/Object@0x1060FFC8[fat lock]
   at test/Deadlock.loopForever(Deadlock.java:67)@0x412304FC
   at test/Deadlock$LockerThread.run(Deadlock.java:57)@0x4123042E
   ^-- Holding lock: java/lang/Object@0x105BDCC0[recursive]
   ^-- Holding lock: java/lang/Object@0x105BDCC0[thin lock]
   at java/lang/Thread.startThreadFromVM(Thread.java:1690)@0x411E5F73
   --- End of stack trace
```

Detecting Deadlocks

After the normal stack dumps, BEA JRockit performs a deadlock detection. This is done by finding “lock chains” in the Java application. If a lock chain is found to be circular, the application is considered caught in a deadlock.

What is a “Lock Chain”?

Although they appear somewhat complex, lock chains are fairly straightforward; they can be defined as follows:

- Threads A and B form a lock chain if Thread A holds a lock that Thread B is trying to take. If A is not trying to take a lock, then the lock chain is “open.”
- If A->B is a lock chain, and B->C is a lock chain, then A->B->C is a more complete lock chain.

- If a Thread D doesn't exist, meaning lock chain C->D doesn't exist, then A->B->C is a complete and open lock chain.

Lock Chain Types

BEA JRockit analyzes the threads and forms complete lock chains. There are three possible kinds of lock chains: Open, Deadlock and Closed lock chains.

Open Chains

Open lock chains represent a straight dependency, as described in What is a "Lock Chain"?. Thread A is waiting for B which is waiting for C, and so on.

Deadlock Chains

Deadlock (circular) chains are similar to an open lock chain, except that the first element is waiting for the last element, in the simplest case: A is waiting for B, which is waiting for A. Note that a deadlocked chain has no head. BEA JRockit selects an arbitrary thread to display as the first element in the chain.

Closed Chains

Closed chains are like open chains, but the first element in the chain is waiting for a lock in another chain. This other chain may be open, deadlocked or closed. If the other chain is deadlocked, then the closed chain is also deadlocked. Note that the division between a closed chain and the other chain is arbitrary.

Closed chains arise whenever two different threads are blocked trying to take the same lock; for example: Thread A holds lock Lock A while Thread B is waiting for Lock A; Thread C is also waiting for Lock A. BEA JRockit will interpret this in one of the following ways:

- B > A as an open lock chain.
- C > A as a closed lock chain.
- C > A as an open lock chain.
- B > A as a closed lock chain.

The only item you might find of interest is if you have a deadlocked lockchain. This can never be resolved, and the application will be stuck waiting indefinitely. Also, if you have long (but open) lock chains, your application might be spending unnecessary time waiting for locks.

Index