BEA JRockit® Mission Control™

Oracle JRockit Runtime Analyzer

Mission Control 3.0.2
Document Revised: June, 2008

Contents

CHAPTER 1

Welcome to the Memory Leak Detector

The Memory Leak Detector, a component of Oracle JRockit Mission Control, detects memory leaks within Java applications running on the Oracle JRockit JVM. A memory leak means application code is holding on to memory that is not used by the application any more. The Memory Leak Detector is a real-time profiling tool providing information about what type of objects are allocated, how many, of what size, and how they relate to each other. Unlike other similar tools, there is no need to create full heap dumps that you need to analyze at a later stage. The data presented is fetched directly from the running JVM, which can continue to run with a relatively small overhead. When the analysis is done, the tool can be disconnected and the JVM will run at full speed again. This makes the tool viable for use in a production environment.

The purpose of the Memory Leak Detector is to display memory leaking object types (that is, classes) and provide help to track the source of the problem. Another purpose of this tool is to help increase the understanding and knowledge to avoid similar programming errors in future projects.

This help is divided into the following topics:

- Investigating an Object Type
- Investigating an Object Instance
- Viewing Allocation Stack Traces
- Customizing Settings

CHAPTER 2

Getting Started with Memory Leak Detection

Finding memory leaks in an application is a bit of a detective's work. You might not always see the obvious leaks. The Memory Leak Detector is a tool that can help you to locate memory leaks in your application. Once you have found a leaking method, it is a much easier task to fix you code.

This section describes the memory leak detection procedure that you can easily adapt to your normal development environment, through the use of Oracle JRockit Mission Control.

This section is divided into the following topics:

Spotting a Memory Leak

There are several ways to spot a memory leak. Most often the memory leak is spotted after the fact, i.e. you get an `OutOfMemoryError` in your application. This can typically happen in the production environment where debugging possibilities are minimal. This online help will describe two common ways to spot a leak.

When you have spotted a memory leak, you can start drilling down to the real cause of the leak by using the Memory Leak Detector plug-in provided with JRockit Mission Control.

The topics are covered to spot a memory leak:

- Verbose Output
- Management Console Heap Overview
- JRA Recording—the GC Events Tab
- Starting the Memory Leak Detector

Verbose Output

One of the most widely used ways to monitor the activity of the garbage collector is to start the JRockit JVM with the `-Xverbose:gc` option and then watch the output for a while (see Listing 2-1).

Listing 2-1 Example of a `-Xverbose:gc` output.

```
[memory ] 2.703-2.729: GC 262144K->3904K (262144K), 25.857 ms
[memory ] 3.901-3.940: GC 262144K->10820K (262144K), 38.835 ms
[memory ] 4.857-4.913: GC 262144K->19606K (262144K), 56.011 ms
[memory ] 5.780-5.878: GC 262144K->28424K (262144K), 97.406 ms
```

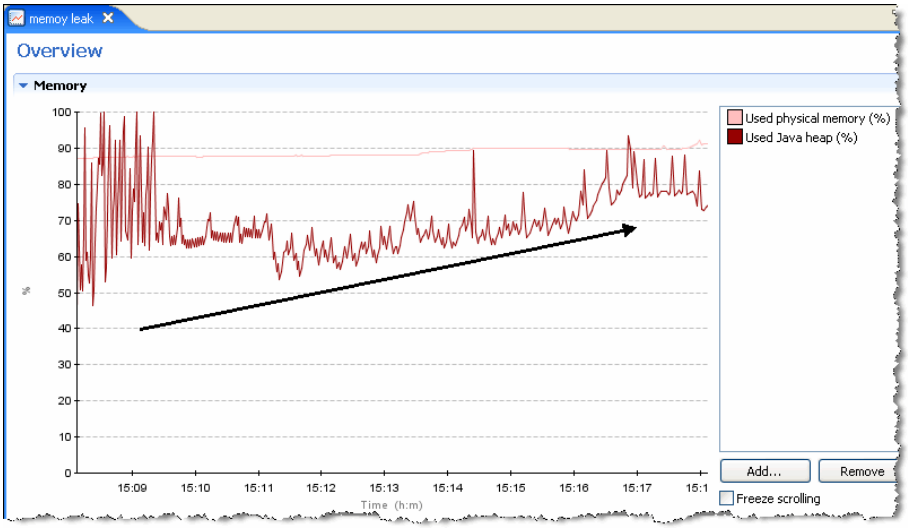
The value after the arrow (`->`) is the heap usage after the garbage collection. You can clearly see that the heap is constantly growing in the example of Listing 2-1.

Management Console Heap Overview

Instead of having to view verbose outputs, tedious at best, you can use the Management Console Overview function to spot a memory leak. In the Management Console's Memory graph you easily see trends in the heap usage over time. Once you have determined that you have a memory leak, you should switch to the Memory Leak Detector to actually find the object that is leaking.

Figure 2-1 shows how a memory leak can look in the Management Console tool.

Figure 2-1 The Management Console Heap Usage Graph

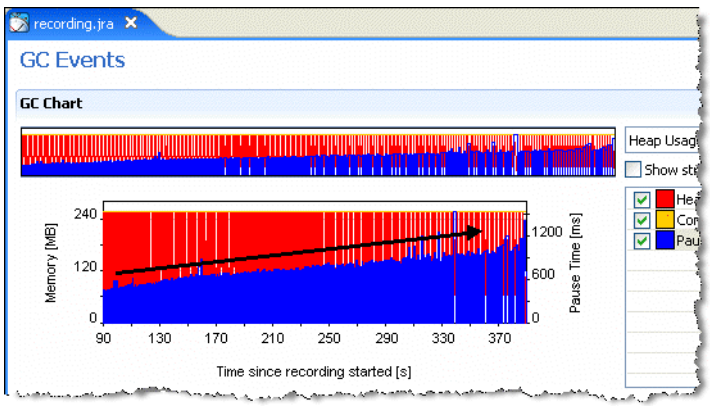


JRA Recording—the GC Events Tab

You can also create a recording of your application and watch the output in the JRA tool—the **GC Events** tab.

Figure 2-2 shows how a memory leak can look in the JRA tool.

Figure 2-2 The JRA Tool with GC Events tab



Now you have determined that you have a memory leak in your system. This means that you can start the Memory Leak Detector to figure out which object is leaking.

Starting the Memory Leak Detector

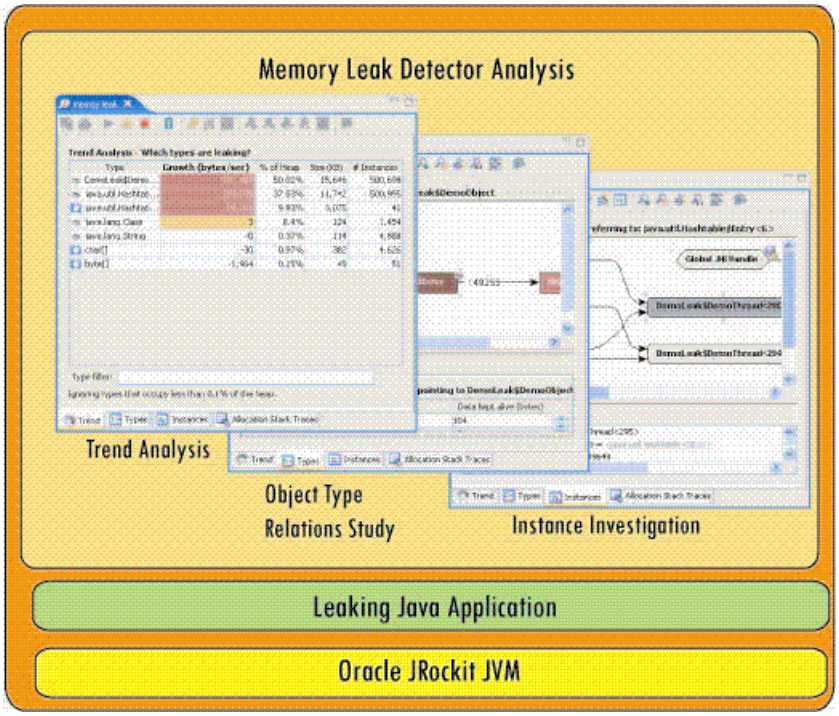
1. Select a JRockit JVM instance in the JVM Browser.
2. Click on the Memory Leak Connection button.

   The Memory Leak Detector launches in the JRockit Mission Control Client (see Figure 3-1).

Detection Process Used with Memory Leak Detector

Ok, now you have spotted that you have a memory leak. It is now time to start using the Memory Leak Detector to drill down to the cause of the leak. Figure 2-3 gives an overview of the different steps that you need to take to get to your memory leak.

Figure 2-3 Memory Leak Detection Process



1. Trend Analysis—What Data is Leaking?
2. Object Type Relations Study
3. Instance Investigation

Trend Analysis—What Data is Leaking?

Trend analysis means to observe continuously updated object type related information and try to discover object types with suspicious memory growth. These object types should then be studied in the next phase of the memory leak detection process. The information in the trend analysis table is updated every ten seconds or more often if there are very frequent garbage collections.

Note: The trend analysis is a great way to find even the smallest leaks, before they even throw OutOfMemory exceptions.

Object Type Relations Study

Studying **object type relations** means following reference paths between object types. The goal is to find interesting connections between growing object types and what types of objects point to them. Finding the object type guilty of unusual memory growth will lead to the third and final phase of the memory leak detection process.

Instance Investigation

Instance investigation consists of finding an instance of abnormal memory size or an abnormal amount of references being held and then inspecting that instance. When inspecting an instance, values will be displayed; e.g. field names, field types, and field values. These values will hopefully lead you to the correct place for the error in the application code; i.e. where that particular instance of that particular object type is allocated, modified, or removed from the collection, depending upon what the situation implies. Minimizing the problem areas of the ones connected to the suspected instance will most likely lead you on the right track to finding the actual problem causing the memory leak and you will be able to fix it.

CHAPTER 3

Using the Memory Leak Detector

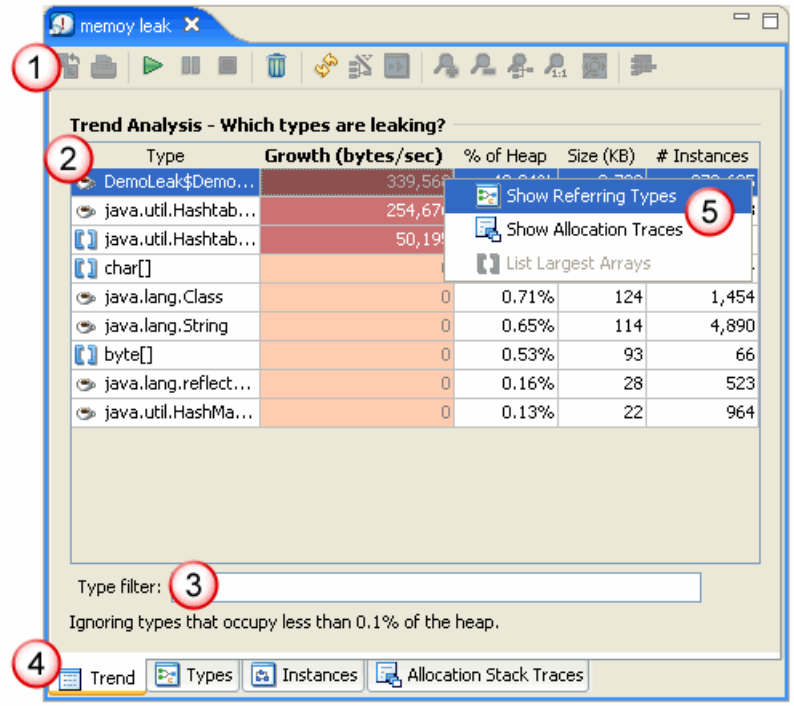
Now you understand how a flow of events for memory leak detection works and the basic functions of the user interface, it is time to get to know how powerful the Memory Leak Detector actually is in action. This part of the user guide describes the different tabs of the interface in detail and how the Memory Leak Detector works when monitoring a Java application with a real memory leak. Each tab of the interface will be explained in detail in this section.

This section is divided into the following topics:

Getting Familiar with the Memory Leak Detector Interface

When you have started the Memory Leak Detector, it starts in the **Trend** tab with the **Trend Analysis** activated, see Figure 3-1.

Figure 3-1 The Main Window of the Memory Leak Detector



1. Toolbar where you control the updating of the analysis, how to view the different graphs when drilling down to find the memory leaks, etc.
2. **Trend Analysis** table that lists all types that consume memory. It lists the types in descending order with the type that leaks the most listed at the top.
3. Applies filter to find a type.
4. **Tabs** that indicate work flow for finding your memory leak.
5. Menu to open types, allocation instances, and other functions.

Toolbar Explained

The Memory Leak Detector tool bar, see Figure 3-2, contains, for example, buttons to connect to the JRockit JVM instance. See Table 3-1 for an explanation of the different tools in the toolbar.

Figure 3-2 The Toolbar in the Memory Leak Detector

Table 3-1 Toolbar Icons Explained

Button	Description
	Export and Print graphs. This button allows you to export the displayed graph to a jpg or gif image.
	Start, Pause, and Stop monitoring your Java application.
	Performs garbage collection on the server.
	Refreshes the current view.
	Re-layouts the graph. This button allows you to re-layout the graph and have the selected
	Restarts the graph. This button allows you to restart, i.e. expand, the node that is currently selected.
	Zooms in and out on a type or an instance. These tools help you navigate in the graph.???
	Zoom in on the selected node and resize the graph to 100%.
	Fills the viewing area with the current graph.
	Lets you view the graph in bird's eye view. You can easily pan and re-center the graph from this view.

Tabs Explained

The main window of the Memory Leak Detector contains four tabs as shown in Figure 3-3. Table 3-2 explains what you can do under the different tabs.

Figure 3-3 The Tabs in the Memory Leak Detector (Indicates Work Flow)

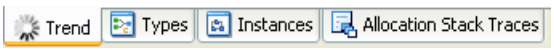


Table 3-2 Memory Leak Detector Tabs Explained

Tab	Description
Trend	From the **Trend** tab you view a trend analysis of the object types on the Java heap. You will see a list of all types that occupy more than 0.1% of the heap (this number can be changed in **File** > **Preferences** > **Trend**). The object type with the highest growth rate will be listed first.
Types	From the **Types** tab you view a type graph that shows how different types point to each other.
Instances	From the **Instances** tab you view an instance graph that shows how different instances point to each other.
Allocation Stack Traces	From the **Allocation Stack Traces** tab you view where a certain type is allocated in the code.

Analyzing the Java Application

From the **Trend tab** (see Figure 3-4), you start the analysis of your applications. The object types with the highest growth in bytes/sec are marked red (darkest) in the **Trend Analysis** table and they are listed at the top of the table. For each update, the list can change and the type that was the highest move down the list. The object types listed in Figure 3-4 are fetched from an example application, where you can suspect a memory leak at the objects marked red.

Figure 3-4 Trend Analysis

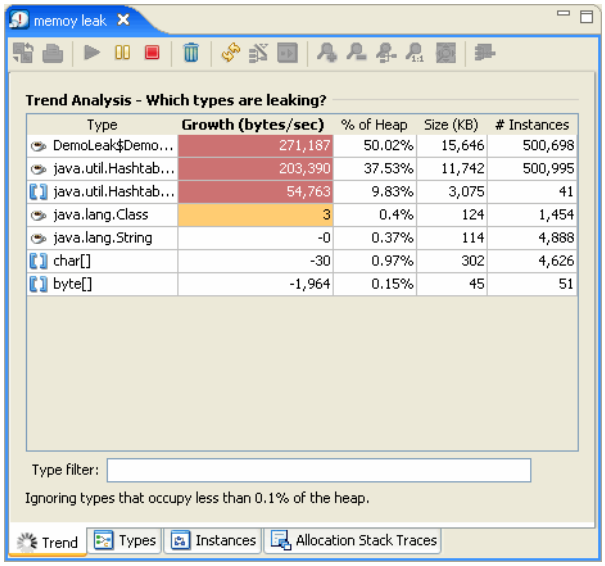


Table 3-3 explains what each column in the **Trend tab** displays.

Table 3-3 Trend Analysis - Which types are leaking?

Column Title	Displays
`Type`	The type of object (class).
`Growth (bytes/sec)`	The amount of memory (in bytes) with which the type is growing, per second.
`% of Heap`	How much of the Java heap is occupied by this type of object, measured in percentages of the entire heap.
`Size (KB)`	What size in KB does that percentage correspond to.
`# Instances`	The number of live objects of this type that currently exist.

Analysis Procedures

This section contains the procedures you can use analyze your application. It shows you how:

- To Start Analyzing Your Application

To Start Analyzing Your Application

- The trend analysis should start automatically. If not, click the **Start** button to start the trend analysis

  If you have an application with a memory leak, the trend analysis can look something like Figure 3-4, where you have a type with a great growth in memory consumption. The darker (red) the type is, the greater amount of memory it consumes.

To Pause Analysis of Your Application

- Click the **Pause** button.

  This operation freezes the updating of the trend analysis in the **Trend tab** and you can start to analyze the application. If you want to view more data from the same analysis run, click the **Play** button again and the Memory Leak Detector resumes displaying samples from the application.

To Stop Analysis of Your Application

- Click the **Stop** button.

  This operation stops the continuous update of the data. When you start the trend analysis again, the data that is currently displayed will be reset.

  Note: You do not stop the application itself by stopping the analysis.

To Start Further Investigation

1. Right-click the object you think contains a memory leak.
2. Select **Show Referring Types**.

   The **Types** tab appears (see Figure 3-5). For instructions on how to investigate further, see To Get Closer to the Memory Leaking Object.

To Jump to Application Source Code

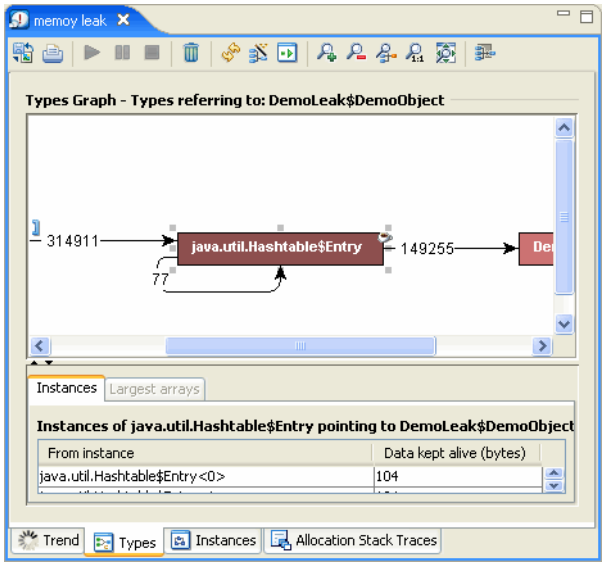
You can jump directly from the Trend Analysis table to application source code. For instructions, please refer to Jumping to Application Source.

Investigating an Object Type

Once you have found a suspected memory leak (a type that is high in growth and is colored red), you investigate the suspected leak further in the **Types tab**, see Figure 3-5. Before anything is displayed in this tab, you need to start the investigation by selecting a type from the **Trend tab**, see To Start Further Investigation, and then right-click on the type and select **Show Referring Types**.

The **Types tab** offers a view of the relationships between all the types pointing to the type you are investigating. For each type you also see a number, which is the number of instances that point to that type.

Figure 3-5 Types Tab



The color red (dark) means that the type has a high growth rate (which may or may not be related to a memory leak).

Object Type Investigation Procedures

This section shows you how:

- To Get Closer to the Memory Leaking Object
- To Get an Overview of the Graph
- To Investigate an Instance of a Type

To Get Closer to the Memory Leaking Object

1. Double-click on the type with the darkest color.

   The type expands further.

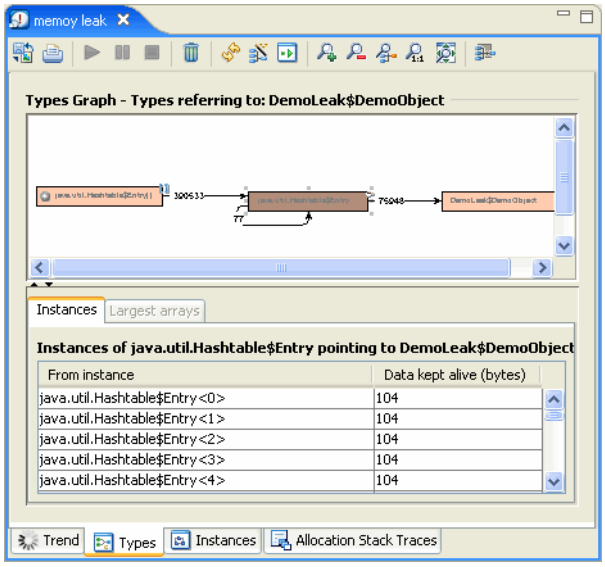
2. Keep clicking the type with the darkest color (red), until you get down to a “natural end” where you think you can pinpoint the memory leak.

3. Right-click the type where you suspect a leak.

4. Select **List Instances**.

   The **Instances** part of the **Types** tab opens.

   List instances shows you instances of the selected type. The instances shown will only be those that have references to the type indicated by the arrow from the selected type in the above type graph.



The lower half of the tab lists all instances of type A pointing to type B if the instance list is not too large. If the list is too large, the Memory Leak Detector might time out when trying to display the list. You can change the time out setting under **Window > Preferences**.

The column **Data kept alive (bytes)** shows how much data a certain instance keeps alive. This data cannot be garbage collected.

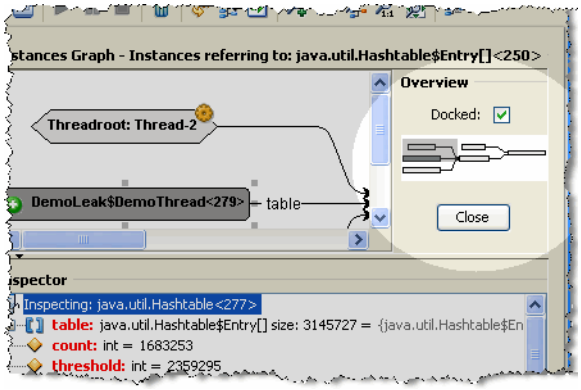
Have the Overview part of the window open to see where you are in the graph (see To Get an Overview of the Graph for how to turn on the overview). You can also zoom in/out or re-center the view.

To Get an Overview of the Graph

- Click the **Bird's-eye Overview** button.

  A small Overview window opens on the tab. This Overview is good to help you navigate in large graphs. You can refocus the view in the current tab by moving the shaded area.

Figure 3-6 Bird's Eye Overview of Graph



To Investigate an Instance of a Type

1. Right-click an instance in the **Types** tab (probably one with the highest data kept alive).
2. Select **Show Referring Instances**.

   The Instances tab appears (see Figure 3-7).

Investigating an Object Instance

In the **Instance tab**, see Figure 3-7, you view the instances of the type that you suspect is leaking memory. You can also see the name of the specific field by looking at the arrow that is referring. Right-click an instance to get a popup menu with the Inspect Instance option. When inspecting an instance you will see all instance variables that the object contains. This information will help you pinpoint where in your application the leaking object is located.

Figure 3-7 Referring Instances Tab

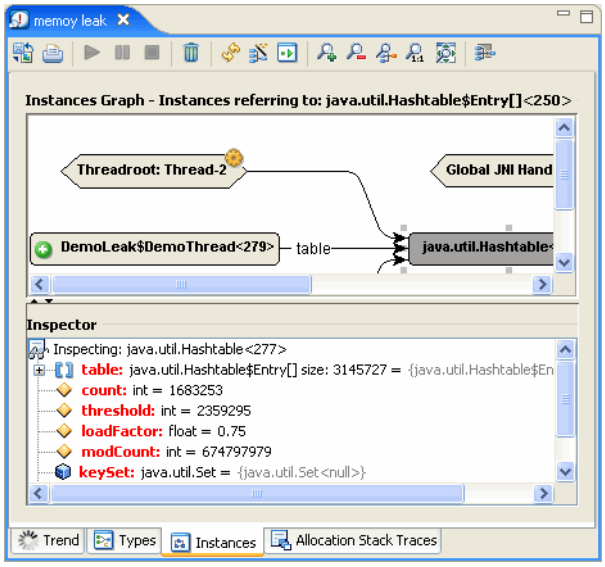


Table 3-4 explains what you will be able to view in the **Instances tab**.

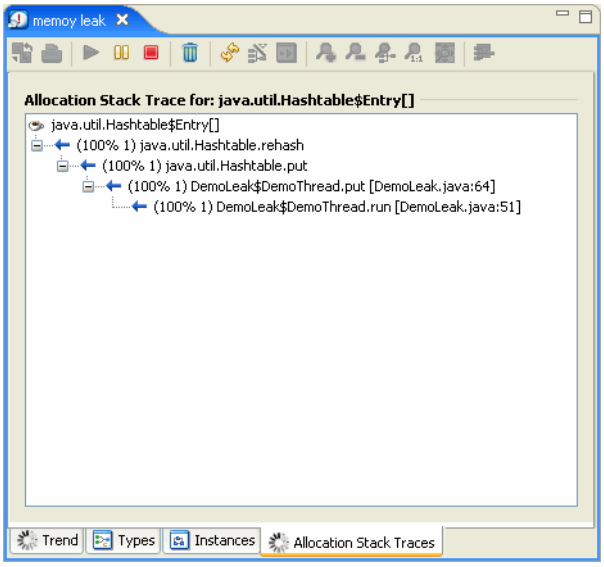
Table 3-4 Instances of Suspected Memory Leaks

Part of Tab	Displays
Instances Graph	This graph shows how the instances are connected to each other.
Inspector	In the inspector view you can see all fields the object contains and their values. The information that is displayed is depending on the application you monitor.

Viewing Allocation Stack Traces

In the **Allocation Stack Traces tab**, see Figure 3-8, you can check for where in the code allocations of a certain type are done. Enabling allocation stack traces may deteriorate the performance of the JRockit JVM. Collecting information about all the allocation points might take a while.

Figure 3-8 Allocation Stack Traces Tab



To Jump to Application Source Code

You can jump directly from the Allocation Stack Trace list to application source code. For instructions, please refer to Jumping to Application Source.

Jumping to Application Source

If you are using the Memory Leak Detector as an Eclipse plug-in, you can jump from either the Trend tab or the Allocation Stack Traces tab directly to the source code. A feature called *Jump-to-Source* allows you not only to see the name of a "problem" class or method displayed in the GUI, but lets you jump from the displayed method or class name directly to that class or method's source, where you can evaluate the code to see what might be causing the problem. This feature extremely is useful in helping you locate and debug coding errors that are creating runtime problems for your application.

To jump to the source code

1. In the **Trend Analysis** table or the **Allocation Stack Trace for...** table, right-click the problem method or class to open a context menu.
2. Select **Open Method** or **Open Type** (depending upon what you are jumping from).

3. The source code appears in a separate editor.

Customizing Settings

You can change preferences for time-outs, communication ports, updating frequency, etc. for the Memory Leak Detector.

Setting Customization Procedures

This section shows you how:

- To Open the Preferences Window
- To Change General Settings
- To Change Communication Settings
- To Change Graphs Settings
- To Change Instance Limits
- To Change Trend Settings

To Open the Preferences Window

- Click **Window > Preferences > JRockit Mission Control > Memory Leak Detector**.

  The **Preferences** window opens.

To Change General Settings

1. Click **Window > Preferences > JRockit Mission Control > Memory Leak Detector**.
2. Deselect the **Attempt to run Memory Leak Detector** in a tab option.

   If you let the Memory Leak Detector open in its own window, it de couples from the JRockit Mission Control window and it opens in a new window. The menus changes slightly to become more similar to the previous release. This setting takes effect the next time you start the Memory Leak Detector.
3. Click **OK**.

To Change Communication Settings

1. Click **Window > Preferences > JRockit Mission Control > Memory Leak Detector** > **Communication**.
2. Select one of the communication options for the Memory Leak Protocol (MLP) communication.
   - **Let OS choose**—this is a convenient choice if you are not sitting behind a firewall.
   - **Use fixed port**—this is used when you are running one JRockit JVM behind a firewall.
   - **Use port relative to JMX port**—this is used when you are running several JVMs on the same computer and the computer is behind a firewall. You decide how many increments (a number between 1 and 65535) you want the TCP port for the MLP communication to be.
3. Click **OK**.

To Change Graphs Settings

1. Click **Window > Preferences > JRockit Mission Control > Memory Leak Detector > Graphs**.

   The following can be set:

   - **Animate layouts**, when selected, animates the expansion of a node on the type in **Types tab** and the **Instances tab**.
   - **Automatically center the last expanded node**, when selected, centers a type in the viewing area for the **Types tab** and **Instances tab**.
   - **Number of referred nodes to add when a node is expanded**, when set, controls how many nodes you want to be displayed in the **Types tab** and **Instances tab**. If you specify a very high number, the view can become cluttered.
   - **Show fully qualified class names in graph nodes**, when selected, displays the complete class name in the graphs of the **Types tab** and **Instances tab**.
2. Click **OK**.

To Change Instance Limits

1. Click **Window > Preferences > JRockit Mission Control > Memory Leak Detector > Instance limits**.

The following can be set:

- **Number of array elements to fetch when inspecting arrays—**These elements are displayed in the **Types** tab when you have selected **List Largest Arrays**.
- **Maximum number of instance relations to list**—Here you set the number of instances you want to list when doing **List instances** of a type. The list is shown in the **Types** tab under **Instances**.
- **Timeout in seconds for fetching instance relations**—The instance relation is showed in the **Instances** tab. A time out error can be caused by too many instances that need to be fetched.
- **Maximum keep alive size to trace, in bytes**—The Memory Leak Detector looks at an object until it reaches this value.

2. Click **OK**.

To Change Trend Settings

1. Click **Window > Preferences > JRockit Mission Control > Memory Leak Detector > Trend**.

   The following can be set:

   - **Lowest heap usage to report**.
   - **Trend refresh interval**.

2. Click **OK**.