



BEA JRockit Memory Leak Detector

**Using the BEA JRockit
Memory Leak Detector**

Copyright

Copyright © 2005 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, BEA JRockit, BEA Liquid Data for WebLogic, BEA WebLogic Server, Built on BEA, Jolt, JoltBeans, SteelThread, Top End, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Manager, BEA MessageQ, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Enterprise Security, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Java Adapter for Mainframe, BEA WebLogic JDriver, BEA WebLogic JRockit, BEA WebLogic Log Central, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server Process Edition, BEA WebLogic WorkGroup Edition, BEA WebLogic Workshop, and Liquid Computing are trademarks of BEA Systems, Inc. BEA Mission Critical Support is a service mark of BEA Systems, Inc. All other company and product names may be the subject of intellectual property rights reserved by third parties.

All other trademarks are the property of their respective companies.

Contents

Introduction

What is New in the BEA JRockit Memory Leak Detector?	1-2
The BEA JRockit Memory Leak Detector Overhead	1-2
About this User Guide	1-2
Finding Additional Information	1-2

Getting Started with Memory Leak Detection

Overview of the Memory Leak Detection Process	2-1
Starting the Memory Leak Detector.	2-2
Touring the Memory Leak Detector Interface	2-2
Tabs Explained	2-3
Toolbar Explained.	2-4
Status Bar Explained.	2-4

Using the Memory Leak Detector

Analyzing the Application	3-1
Investigating a Suspicious Object Type.	3-3
Investigating an Object Instance	3-8
Viewing Allocation Stack Traces	3-9
Customizing Settings	3-10

Help Us Improve JRockit

How will BEA Systems Use This Feedback	4-1
--	-----

BEA JRockit Support for the Memory Leak Detector	4-2
Frequently Asked Questions	4-2
Does BEA Systems Guarantee the Accuracy of this tool's output?	4-2
Does the Memory Leak Detector Cause Any Overhead?	4-2
What Kind of Support is Available for the Memory Leak Detector?	4-2
Is There a Forum Where I can Discuss the Memory Leak Detector?	4-3
Known Issues	4-3

Introduction

This product is provided “as-is,” without any expressed or implied warranties or support by BEA Systems, Inc. This product, which may or may not become an officially supported product from BEA Systems, may contain errors and/or inaccuracies. Use of this product is left solely upon the discretion of the user without any endorsement from BEA Systems. The Memory Leak Detector functionality may or may not be available in future BEA JRockit versions. Questions and problems may be reported via online BEA JRockit newsgroups at <http://newsgroups.bea.com>.

The BEA JRockit Memory Leak Detector detects memory leaks within Java applications that run on BEA JRockit. A memory leak means application code is holding on to memory that is not used by the application any more. The BEA JRockit Memory Leak Detector is a real-time profiling tool that gives 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 this tool 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.

Note: To access the full version of the BEA JRockit Memory Leak Detector, JRockit J2SE 5.0 sp1 or higher is required.

The following subjects are covered in this section:

- [What is New in the BEA JRockit Memory Leak Detector?](#)
- [The BEA JRockit Memory Leak Detector Overhead](#)
- [About this User Guide](#)
- [Finding Additional Information](#)

What is New in the BEA JRockit Memory Leak Detector?

- The biggest change from the previous version of the Memory Leak Detector for BEA JRockit 1.4.2_05 is that it is now a stand-alone tool and you do not need to access it through the BEA JRockit Management Console.
- Major improvements on usability and look and feel have also been done.

The BEA JRockit Memory Leak Detector Overhead

The extra cost of running the BEA JRockit Memory Leak Detector against a running BEA JRockit is very small and is noticeable only in that garbage collections take a little bit more time. The overhead of enabling allocation stack traces can be more significant and should therefore be used with care. This provides for a low cost monitoring and profiling of your application.

About this User Guide

In this document you will be guided through how you can spot a memory leak in your Java application. You will also get hints on how to repair a memory leak. This user guide assumes that you know what a JVM is and that you are familiar with Java application development.

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 Online Documentation](#).

Getting Started with Memory Leak Detection

This section describes the BEA JRockit Memory Leak Detector (from now on referred to as Memory Leak Detector) start-up procedure and the user interface. Information on the following topics are included:

- [Overview of the Memory Leak Detection Process](#)
- [Starting the Memory Leak Detector](#)
- [Touring the Memory Leak Detector Interface](#)

Overview of the Memory Leak Detection Process

The memory leak detection process consists of three phases:

1. Trend analysis
2. Object type relations study
3. Instance investigation

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.

Studying **object type relations** means repeatedly 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 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.

Starting the Memory Leak Detector

Before you start the Memory Leak Detector and your application, you need to start the management server.

1. Start your Java application with the BEA JRockit JVM as usual, but add the `-Xmanagement` option to the command line.
2. Start the Memory Leak Detector by typing `java -jar MemoryLeakDetector.jar` in a command window.
3. Enter a name for the server in **Server name**. This is the name (or IP address) of the computer that runs JRockit and the application that you want to monitor.

The default port is 7091. For changing the port of the management server, see “[Changing the Port](#)” in [Using the BEA JRockit Management Console](#) at:

<http://edocs.bea.com/wljrockit/docs50/usingJMC/start.html#1036322>

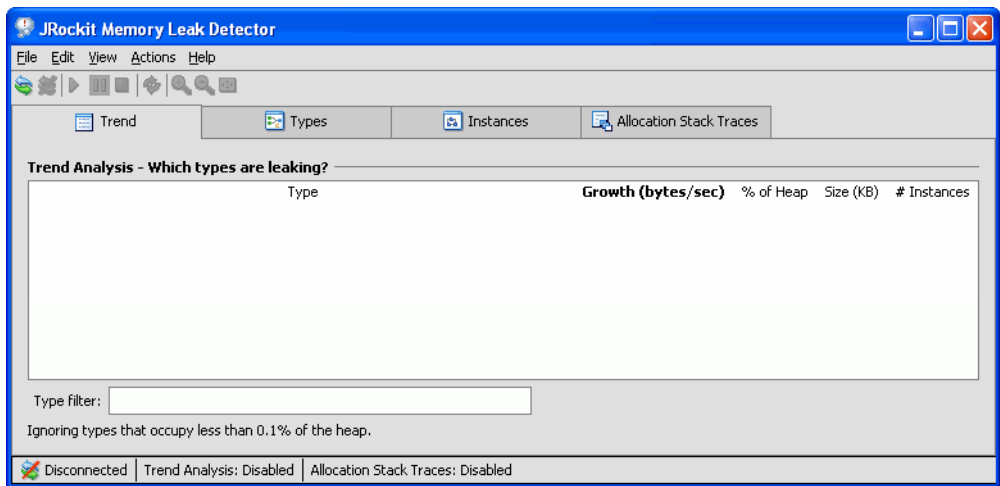
4. Click **Connect** in the **Connect to JRockit** window.

The Memory Leak Detector window opens (see [Figure 2-1](#)).

Touring the Memory Leak Detector Interface

When it is not connected to any BEA JRockit JVM, the Memory Leak Detector window looks like [Figure 2-1](#). The interface consists of four tabs, a tool bar, main menus, and a status bar.

Figure 2-1 The Main Window of the Memory Leak Detector







Tabs Explained

The main window of the Memory Leak Detector contains four tabs as shown in [Figure 2-2](#): **Trend**, **Types**, **Instances**, and **Allocation Stack Traces**.

Figure 2-2 The Tabs in the Memory Leak Detector (Indicates Work Flow)



Table 2-1 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. 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.










Toolbar Explained

The Memory Leak Detector tool bar, see [Figure 2-3](#), contains, for example, buttons to connect to the JRockit instance. See [Table 2-2](#) for an explanation of the different tools in the tool bar.

Figure 2-3 The Toolbar in the Memory Leak Detector



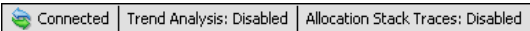
Table 2-2 Toolbar Explained

Icon	Description
	Connect to the management server. This button connects you to the management server, which in turn allows you to monitor your Java application.
	Disconnect from the management server and your Java application.
	Start monitoring your Java application.
	Pause the screen updating.
	Stop the current monitoring.
	Refresh the current view.
	Zoom in on a type or an instance. This tool helps you navigate in the graph.
	Zoom out from a type or an instance.
	Center objects in your viewing area.

Status Bar Explained

The status bar ([Figure 2-4](#)) at the bottom of the window displays information regarding the current connection, whether the trend analysis is on or not, and whether the allocation stack trace is on or not.

Figure 2-4 The Status bar in the Memory Leak Detector



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.

The following topics will be covered in this section:

- [Analyzing the Application](#)
- [Investigating a Suspicious Object Type](#)
- [Investigating an Object Instance](#)
- [Viewing Allocation Stack Traces](#)

Analyzing the Application

From the **Trend tab** (see [Figure 3-1](#)), 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 3-1](#) are fetched from an example application, where you can suspect a memory leak at the objects marked red.

Figure 3-1 Memory Leak Analysis

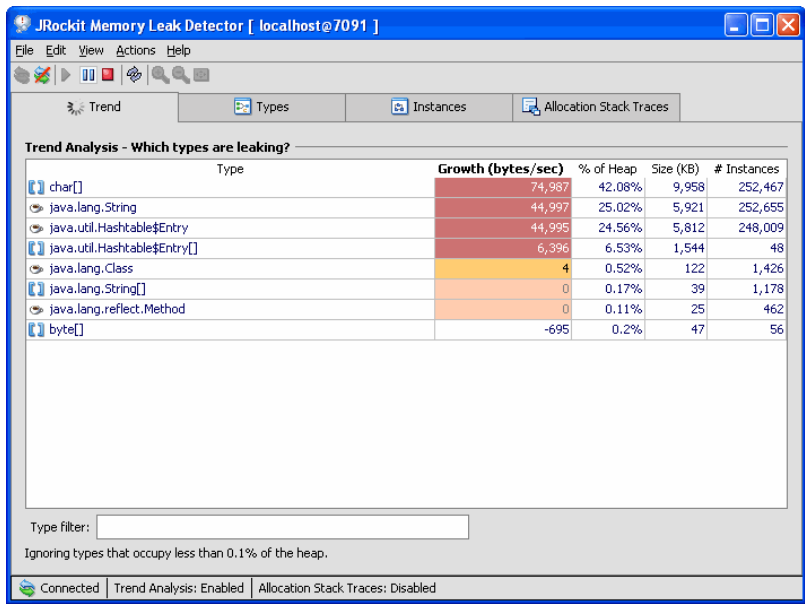


Table 3-1 explains what each column in the **Trend** tab displays.

Table 3-1 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.

To Start Analyzing Your Application

- 1. Make sure the Memory Leak Detector is connected to JRockit and that your JRockit application is running with the `-Xmanagement` option turned on.

2. 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 3-1](#).

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 collect more data from the same sample, click the **Play** button again and the Memory Leak Detector displays the last samples from the application.

To Stop Analysis of Your Application

- Click the **Stop** button.

This operation stops the continuous update of the data and 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 the Investigation

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

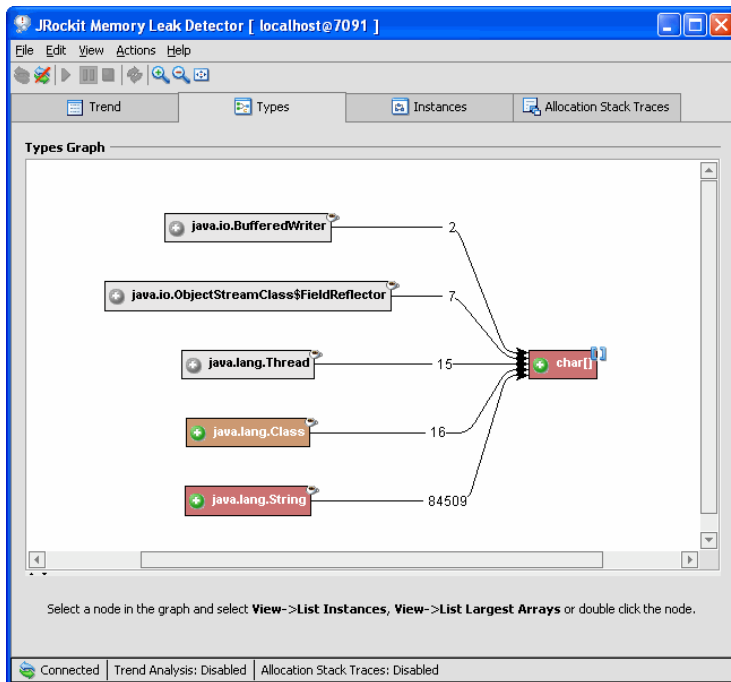
The Types tab appears (see [Figure 3-2](#)). For instructions on how to investigate further, see [To Get Closer to the Memory Leaking Object](#).

Investigating a Suspicious 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 3-2](#). Before anything is displayed in this tab, you need to start the investigation by selecting a type from the **Trend tab**, see [To Start the Investigation](#).

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 3-2 Types Tab



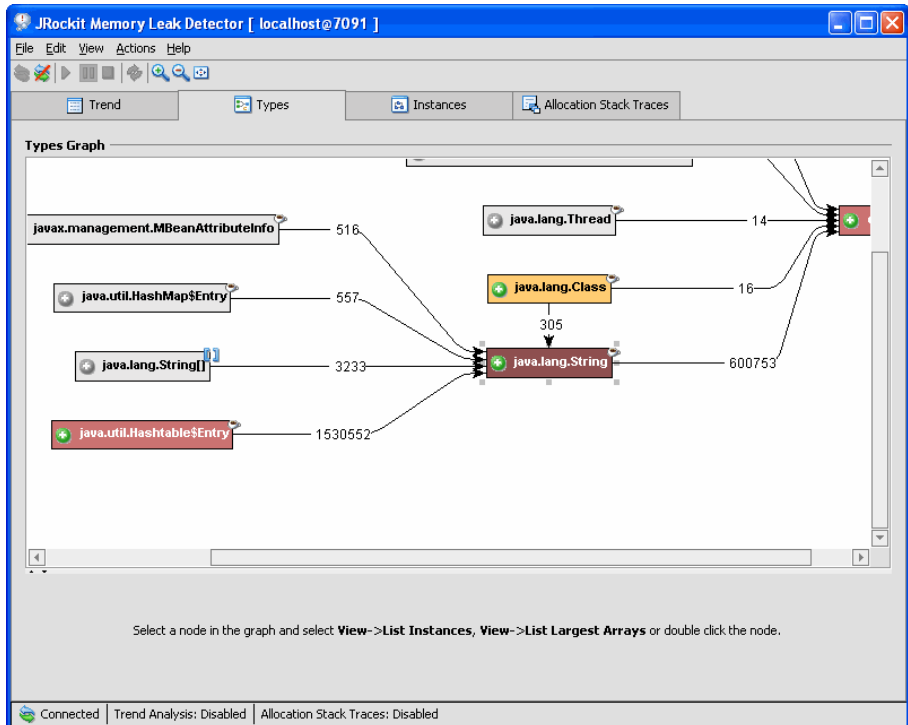
The color red (dark) means that the type has a high growth rate (which may or may not be related to a memory leak).

To Get Closer to the Memory Leaking Object

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

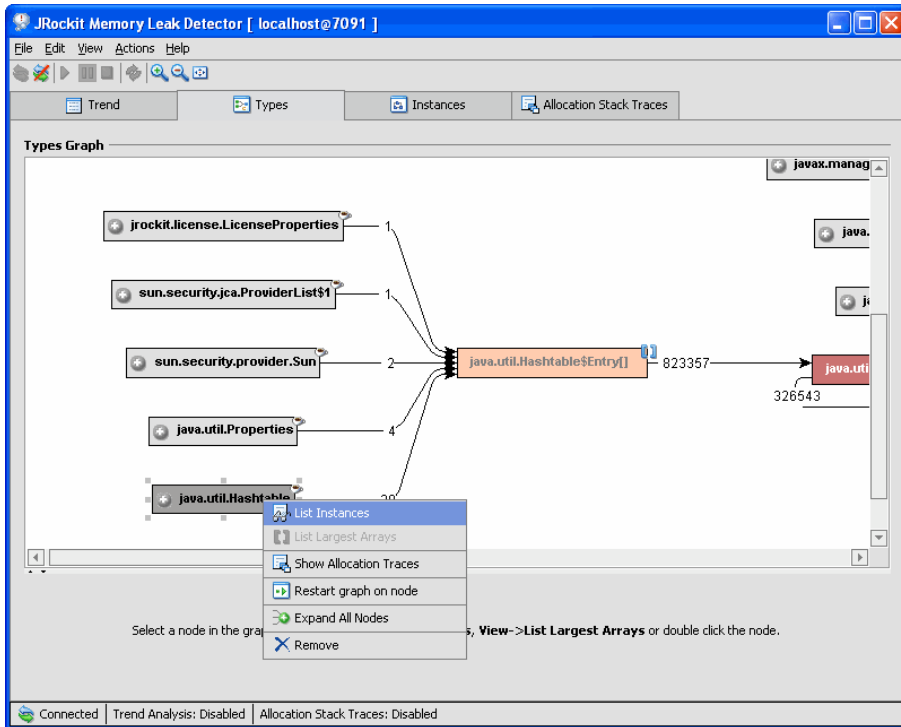
The type expands further (see [Figure 3-3](#)).

Figure 3-3 Type Graph Expanded



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 (see [Figure 3-4](#)).

Figure 3-4 Type Graph with Memory Leak Pinpointed

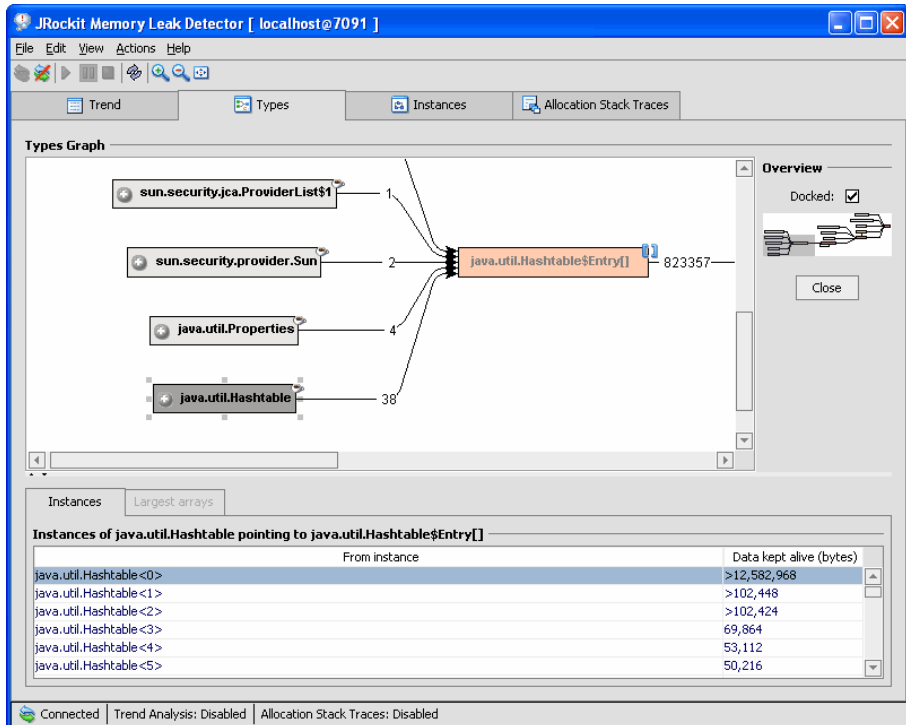


4. Select **List Instances**.

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.

Figure 3-5 List of Instances in Types Tab



The lower half of the tab lists all instances of type A pointing to type B if the instance list is not too large (see [Figure 3-5](#)). 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 **File > Preferences**.

The column **Data kept alive (bytes)** shows how much data a certain instance keeps alive. That 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 (see [Table 2-2](#) for an explanation of the zooming tools).

To Get an Overview of the Graph

- Click **View > Birds-eye Overview**.

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.

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 3-6](#)).

Investigating an Object Instance

In the **Instance** tab, see [Figure 3-6](#), 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 3-6 Referring Instances Tab

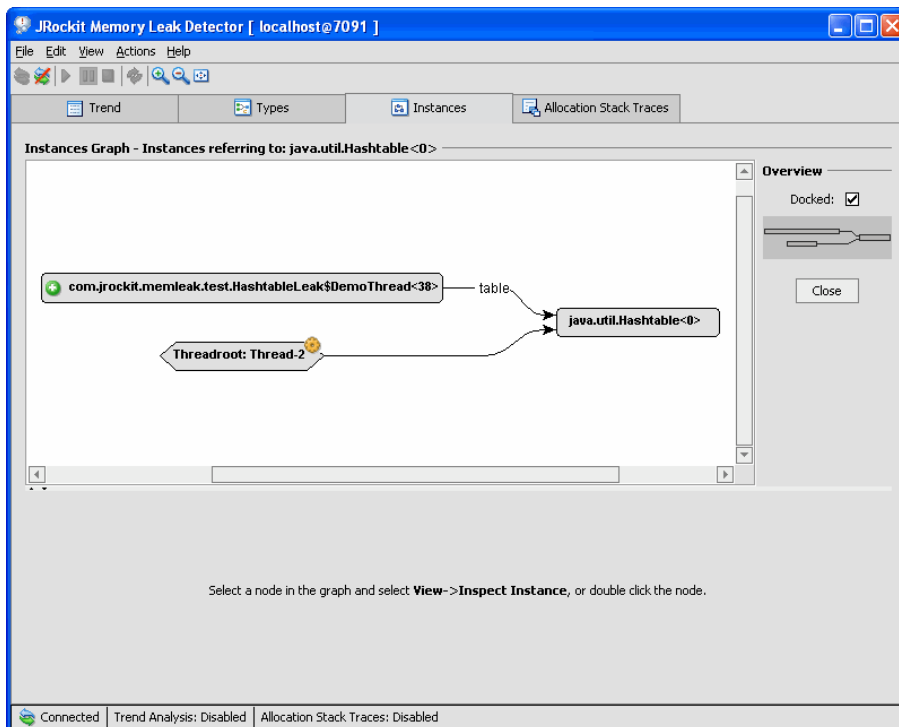


Table 3-2 explains what you will be able to view in the **Instances** tab.

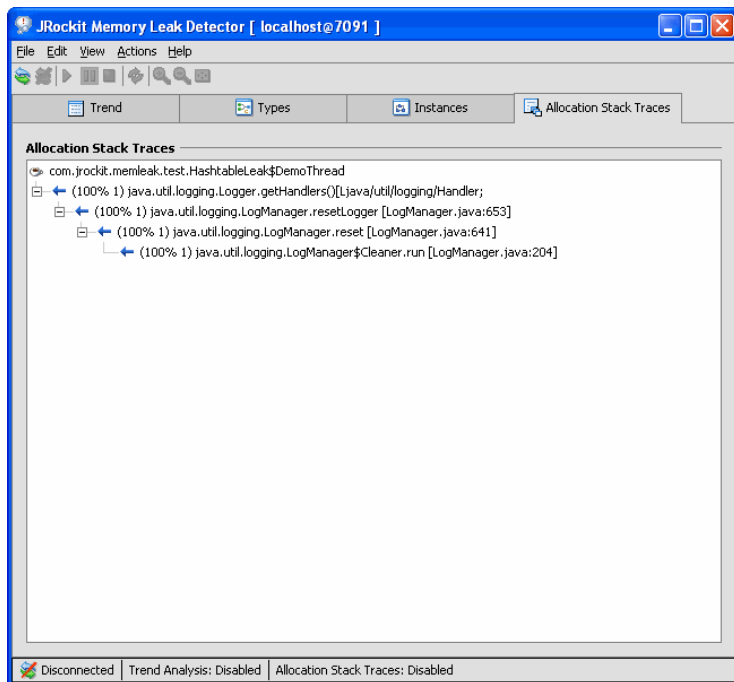
Table 3-2 Instances of Suspected Memory Leaks

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 3-7, you can check for where in the code allocations of a certain type are done. Enabling allocation stack traces may deteriorate the performance of JRockit. Collecting information about all the allocation points might take a while.

Figure 3-7 Allocation Stack Traces Tab



Customizing Settings

The Memory Leak Detector can be customized in many different ways. [Figure 3-8](#) through [Figure 3-13](#) explains the different settings you can make.

To Open the Preferences Window

- Click **File > Preferences**.

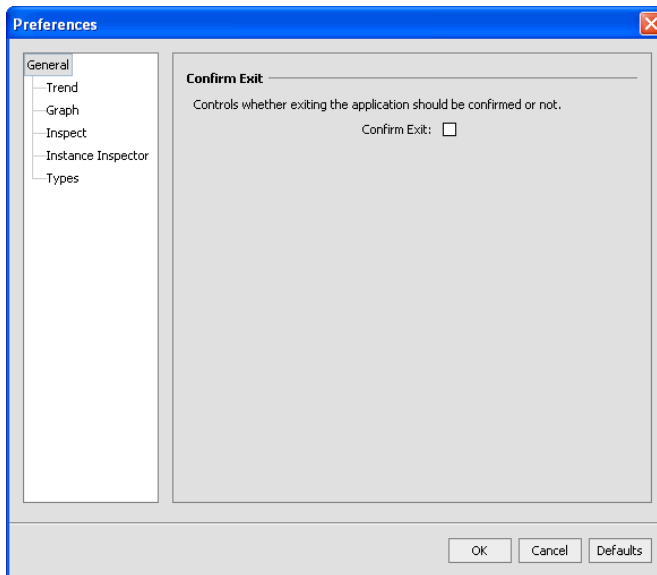
The **Preferences** window opens (see [Figure 3-8](#)).

When you have set your preferences, click **OK** for them to take effect.

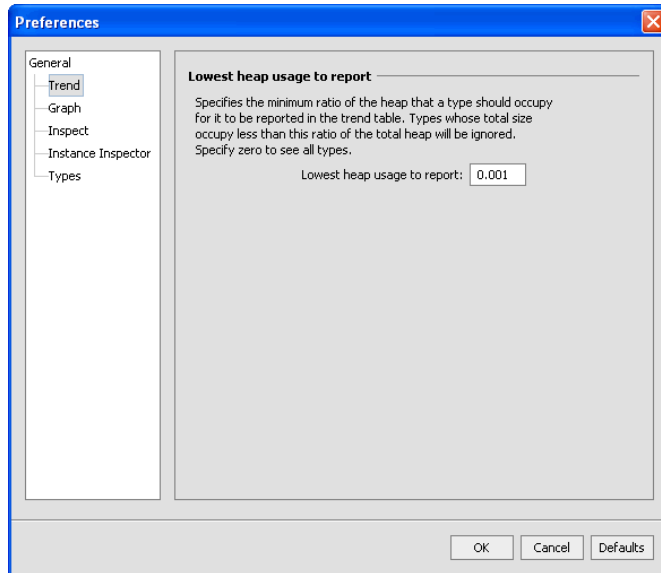
To Reset Preferences to Default Values

1. Click **File > Preferences**.
2. Click **Defaults**.
3. Click **OK**.

Figure 3-8 Confirm Exit Setting

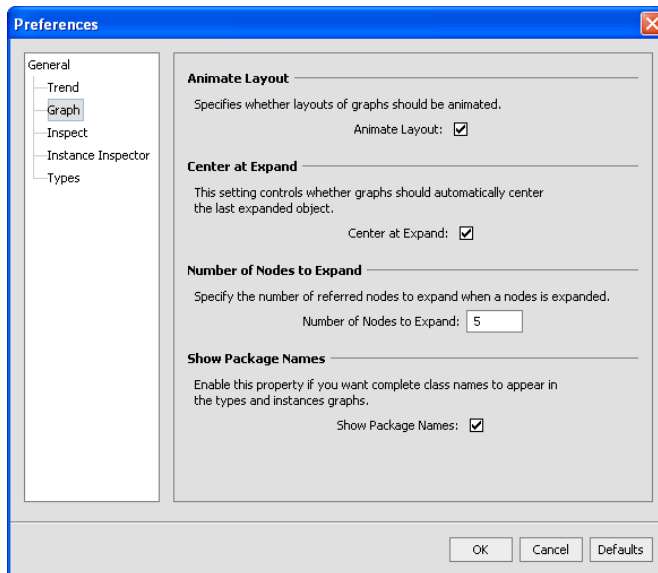


Select the **Confirm Exit** option if you want to get a confirmation message when you are closing the Memory Leak Detector.

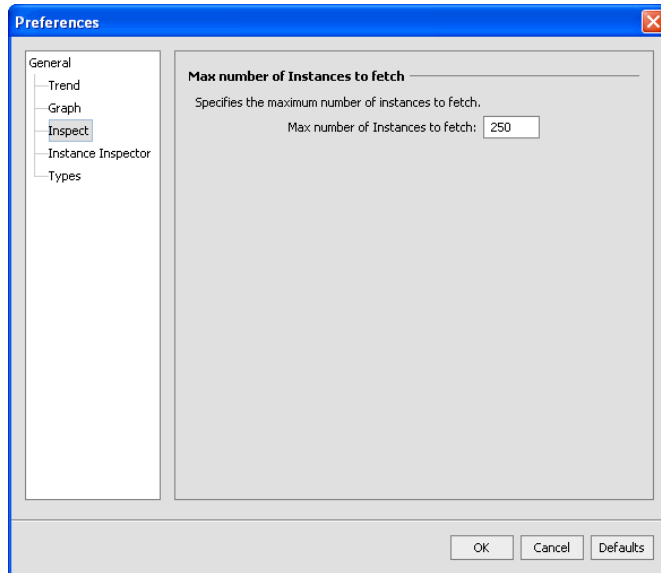
Figure 3-9 Display of Heap Usage

Here you set the ratio of the Java heap that you want displayed. Those types that are smaller than the set ratio are not displayed. If you want to display all types, set the ratio to 0 (zero).

Figure 3-10 Graph Settings

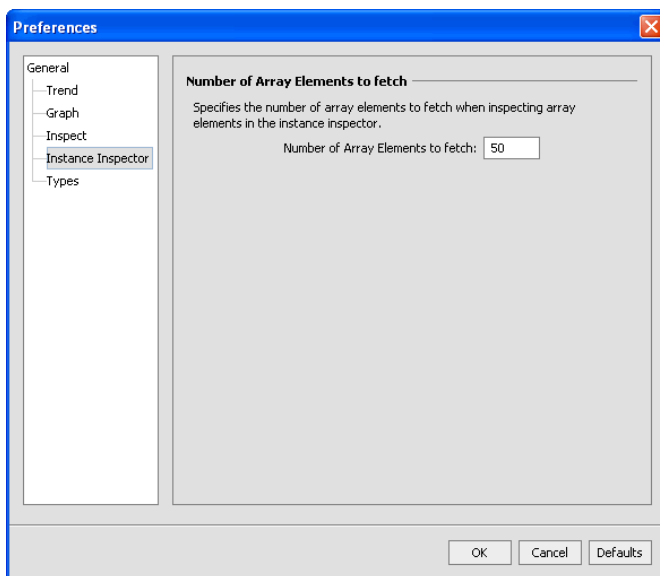


- Select **Animate Layout** if you want the types graph in the **Types tab** and the **Instances tab** to be animated when you expand a node on the type.
- Select **Center at Expand** if you want the graphs in the **Types tab** and **Instances tab** to be centered in the viewing area when you expand a node on the type or instance.
- Specify the value for **Number of Nodes to Expand** to control 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.
- Select **Show Package Names** if you want the complete class name displayed.

Figure 3-11 Number of Instances that are Fetched

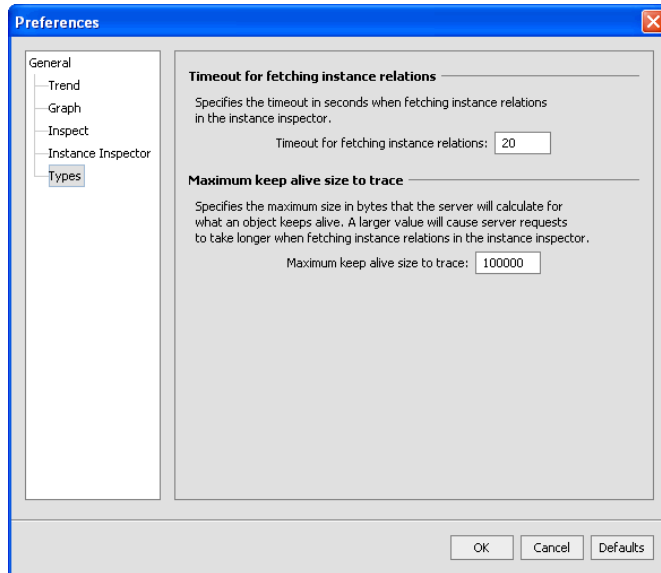
Here you set the number of instances you want to list when doing **List instances** of a type. The list is shown in the **Types tab** under **Instances**.

Figure 3-12 Number of Array Elements that are Fetched



Specify the value for the **Max number of Array Elements to fetch**. These elements are displayed in the **Types tab** when you have selected **List Largest Arrays**.

Figure 3-13 Timeout Setting



- Set a number for **Timeout for fetching instance relations**. This number is measured in seconds. The instance relation is showed in the **Instances tab**. The time out error can be caused by too many instances that need to be fetched.
- Set a value, in bytes, for the **Maximum keep alive size to trace**. When the Memory Leak Detector looks at an object until it reaches this value.

Using the Memory Leak Detector

Help Us Improve JRockit

The Memory Leak Detector is designed to help developers more easily find memory leaks and to better understand critical points of program engineering. It provides an easy way to capture information about object type allocation statistics.

If you have any suggestions about how to improve this tool or information on how it is most commonly used in development environments, we would be grateful to receive your input. This information would contribute to our understanding on how to best further improve this tool in the future.

Please, send an email with feedback and your ideas on how to use it to:

jrockit-improve@bea.com

How will BEA Systems Use This Feedback

The feedback will be considered by the development team designing the Memory Leak Detector. We will look at collected ideas and improve the tools of BEA JRockit to make them even easier to use. Our goal with the development of this tool is to simplify the difficult task of finding memory leaks and help developers work more efficiently.

BEA JRockit is already providing a lot of appreciated manageability tools and, to keep a close dialogue with developers using Java Runtime Environments, BEA Systems is always trying to find ways to improve BEA JRockit. This is one of the ways.

BEA JRockit Support for the Memory Leak Detector

Only more recent versions of BEA JRockit fully support the stand-alone Memory Leak Detector: BEA JRockit 5.0 sp1 and sp2.

Frequently Asked Questions

Following are some questions we have frequently been asked about the Memory Leak Detector:

- [Does BEA Systems Guarantee the Accuracy of this tool's output?](#)
- [Does the Memory Leak Detector Cause Any Overhead?](#)
- [What Kind of Support is Available for the Memory Leak Detector?](#)
- [Is There a Forum Where I can Discuss the Memory Leak Detector?](#)

Does BEA Systems Guarantee the Accuracy of this tool's output?

Since this is not a supported product, we cannot make any guarantees about the accuracy of the data shown or the stability of JRockit when using the Memory Leak Detector.

Does the Memory Leak Detector Cause Any Overhead?

In the trend analysis of the memory leak detection process, the data presented is continuously updated; however, the overhead during this phase is very small. Each garbage collection will take a bit longer. During the second and third phase the only overhead will be some additional garbage collections, which in most cases is negligible. Overall, there is practically no overhead and it should not affect the speed or results of your application.

What Kind of Support is Available for the Memory Leak Detector?

The Memory Leak Detector functionality is currently being provided as-is for your convenience and to help with memory leak detection and is not supported by BEA Support.

Is There a Forum Where I can Discuss the Memory Leak Detector?

If you have any questions you are welcome to share them in the BEA JRockit general interest newsgroup, which is monitored by our engineering team. To access the newsgroup, go to:

<http://newsgroups.bea.com>

Known Issues

Sometimes static fields and the number of thread roots are not correctly displayed in the window displaying instances referring to an other instance. You can fix this by starting the memory leak detection process once again (that is, unfreezing and freezing the memory leak system); however, any data displayed is correct.

Help Us Improve JRockit

Index

A

- allocation stack traces tab 2-3
- animate Layout 3-12
- array elements 3-14

C

- center at expand 3-12
- center objects 2-4
- confirmation message 3-10
- connect 2-4

D

- default port 2-2
- disconnect 2-4

G

- graphs 3-12

H

- heap usage 3-11

I

- instance investigation 2-2
- instance tab 2-3
- instances 3-13

J

- java -jar memleakapp.jar 2-2

L

- list largest arrays 3-14

M

- max number of array elements to fetch 3-14
- maximum keep alive size to trace 3-15

N

- number of nodes to expand 3-12

O

- object type relations 2-1
- overview 3-7

P

- pause 2-4

R

- refresh 2-4

S

- server name 2-2
- show package names 3-12
- start monitoring 2-4
- status bar 2-4
- stop monitoring 2-4

T

timeout for fetching instance relations 3-15
trend analysis 2-1
trend tab 2-3
types tab 2-3

X

-Xmanagement 2-2

Z

zoom in 2-4, 3-7
zoom out 2-4, 3-7