J Optimizer 1.0 User Guide

Embarcadero Technologies, Inc.
100 California Street, 12th Floor
San Francisco, CA 94111

J Optimizer User Guide

Contents

Welcome to **Stand-alone J Optimizer 2009**. Stand-alone J Optimizer provides full profiling cability without requiring integration into an IDE or another, larger development environment. **Touchpoint J Optimizer** is an optional integration plug-in that is packaged with Stand-alone J Optimizer and enables developers to use the J Optimizer profiling tools from within their own Eclipse-based application or environment. Touchpoint J Optimizer does not have a graphical user interface (GUI). You do not need to use Touchpoint J Optimizer in order to use Stand-alone J Optimizer.

In addition, Stand-alone J Optimizer 2009 includes the Source Code Audit and Metrics tool.

1. Overview of J Optimizer

J Optimizer equips Java developers with a comprehensive toolkit for optimizing application performance and quality. You can use **J Optimizer** to profile memory and CPU usage, display real-time threading information, and determine which parts of your code are actually being executed. You can also track performance bottlenecks at the JDBC, JMS, JNDI, JSP, EJB, CCI, and Web Services levels, and locate the exact line of source code for root-cause analysis.

You can profile a number of different Java programs using J Optimizer. For a complete list of the programs you can profile, see the How to Get Started **Help** page.

J Optimizer consists of four profiling tools: **Profiler**, **Code Coverage**, **Thread Debugger**, and **Request Analyzer**. Each tool has its own data-collecting Agent. For example, the Request Analyzer Agent collects data about JEE protocols, while the Code Coverage Agent collects data on code use. You can launch only one profiling tool at a time.

About the J Optimizer Profiling Tools

The table below summarizes the purpose of each profiling tool. For more detailed information about a tool, click on the tool's name in the **Tool** column. This takes you to a group of **Help** topics about that tool.

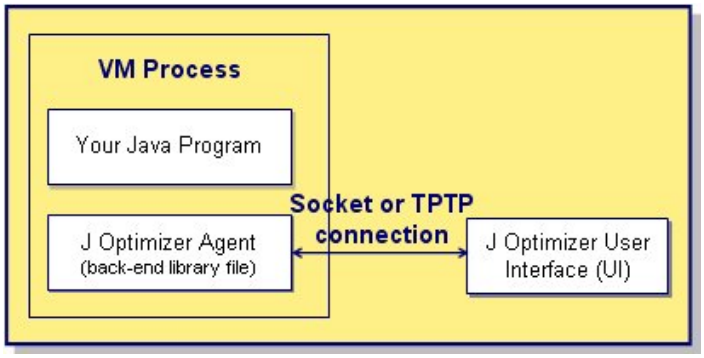
Tool	Purpose
Profiler	Use the **Profiler** to examine memory and CPU use in the target application. The Profiler can identify memory leaks, inefficient temporary-storage issues, CPU bottlenecks, and unit test performance regressions.
Code Coverage	Use **Code Coverage** to to identify and analyze the classes, methods and lines of code that are being executed when your target application runs. You can test applications, applets, JavaBeans, Enterprise JavaBeans (EJBs), JavaServer Pages (JSPs)and virtually any other Java code. J Optimizer to identify and remove dead code, improve quality, and improve the application's footprint.
Thread Debugger	Use the **Thread Debugger** to identify and analysize thread-contention issues, thread-starvation, excessive locking, deadlocks, and other thread-related issues.
Request Analyzer	Use the **Request Analyzer** to carry out a CPU performance analysis of JEE protocols. You can obtain precise drill-down information about performance bottlenecks in any one of JDBC, JNDI, CCI, RMI, EJB, JSP, JMS, or WSVC protocols. This tool also provides protocol-specific quality analysis of unclosed or overused resources, exceptions, and other potential issues.

Note: All J Optimizer Agents can profile JEE applications. However, only the Request Analyzer can collect data for JEE-related events such as the activation or passivation of specific EJBs.

How J Optimizer works with a Java Virtual Machine

The **J Optimizer** profiling tools work with either of the two standard profiling interfaces that a Java Virtual Machine (JVM) might support: the Java Virtual Machine Profiling Interface (JVMPI) or the Java Virtual Machine Tool Interface (JVMTI). The J Optimizer tool collects data from the virtual machine through JVMPI or JVMTI callbacks. You can specify a VM when you create a profile configuration in the J Optimizer user interface, or, if you do not have access to the Optimizer UI, from a command line. J Optimizer can also request specific event notification, based on options you specify.

The figure below illustrates how a J Optimizer profiling agent collects data from within the same VM that is running the Java program, then displays it in the Optimizer UI.



Note: A TPTP connection can only be used with the JBuilder plug-in version of J Optimizer.

Related Topics

Tour of the J Optimizer User Interface (UI)

About Using J Optimizer from a Command Line

1.1. How to Get Started with J Optimizer

Your first step to profiling with J Optimizer is to select a mode of use. You can use J Optimizer in the following ways:

- **From the Graphical User Interface (UI):**

  If you are using the JBuilder plugin or stand-alone versions of J Optimizer, you can use the J Optimizer UI to select the profiling tool you want to use, configure tool options, select the Java program you want to profile, launch the profiling tool, and view profiling results.

  Note: If you are using Touchpoint J Optimizer, you have plugged J Optimizer into an Eclipse application, and can use that application's UI to create a profile configuration to select a tool and start profiling. Alternatively, you can use commands to perform these tasks. To view profiling results, use the Stand-alone J Optimizer UI.

- **From a Command Line:**

  You can use commands to select the profiling tool you want to use, configure its attributes as desired, and start both the Java program to profile and the profiling tool. You must connect to the J Optimizer UI to view profiling results.

Overview of J Optimizer Profiling Tasks

The following table provides an overview of the steps you take to configure and use the J Optimizer profiling tools. For details and instruction on how to perform a step, click on a link in the step descriptions. For more information about each of the profiling tools, see the links in the **Related Topics** section.

Step	From the UI	From a Command Line
1	Use the Profile Configuration Wizard to select: --The Java program to profile. --The profiling tool. --Edit tool options if desired. See your "About Using the Profile Configuration Wizard" **Help** page for more information.	Configure your machine and application or server to work This typically involves setting environmental variables and config file. If you're using a TPTP connection, you'll take a steps.
2	Use the Profile Configuration Wizard to start profiling. See your "About Using the Profile Configuration Wizard" **Help** page for more information.	Select and launch the startup argument and the profiling to
3	View profiling results.	Attach to the J Optimizer UI and view profiling results. Inst socket and TPTP connections are provided.

Types of Java Programs You Can Profile with J Optimizer

The following table describes the Java programs you can profile with J Optimizer:

Java Program Type	Program Description	J Opti for JBuilder	Stand-alone J Opti	Touchpoin
Attach - J Optimizer Agent	Use to profile larger programs such as application servers.	X	X	X
Eclipse Application	Use to profile the Eclipse application and plug-ins that you have created to work with Eclipse.	X	X	X
External Java Application	Use to profile precompiled Java applications.	X	X	X
Java Applet	Use to profile Java applets created within	X	X	X
Java Application	Use to profile Java applications created within	X	X	X
JUnit	Use to profile JUnit tests.	X	X (external only)	X
JUnit Plugin Test	Use to profile JUnit test that are designed to profile Eclipse plugins.	X	X	X

OSGI Framework	Use to launch an OSGI framework and profile an OSGI plug-in.	X	-	X
Web Client	Use to profile a Web Client such as	X	-	-
Web Service	Use to profile a Web Service such as	X	-	-

Related Topics

- Overview of J Optimizer
- About Memory Profiling
- About CPU Profiling
- About Code Coverage
- About Thread Debugger
- About Request Analyzer and JEE Profiling
- About Profiling from a Command Line

1.2. Using J Optimizer from the UI

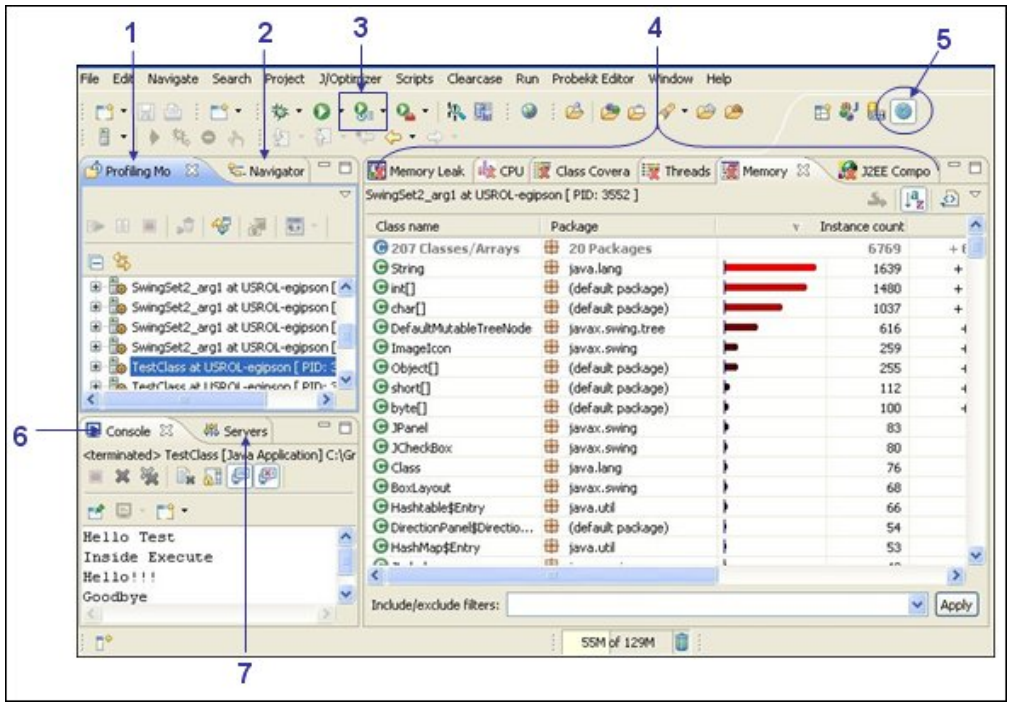
You can use the J Optimizer User Interface (UI) to perform the following profiling tasks:

- Select and edit options for the profiling tool you want to use. You use a Profile Configuration Wizard to select and configure the tool.
- Start profiling with the tool. You also use Profile Configuration Wizard to launch (i.e. start) the profiler.
- View the data collected during the profiling process.

This **Help** topic shows you the UI location from which to launch each task. For instructions on how to perform the same tasks from a command line, click here.

Tour of the J Optimizer UI

The image below identifies the primary elements of the J Optimizer UI. You use these elements to configure and launch a profiling tool, and to view the data collected by each tool. Descriptions of each element follow the image.



1. The Profiling Monitor tab

Until you delete them, the **Profiling Monitor** tab lists all J Optimizer-enabled processes that are running or have been run. Each time you launch a profile configuration, J Optimizer creates a new JVM process and displays the process in the Profiling Monitor. This means that one launch configuration may have several processes displayed in this view. See the "Other UI Elements" section below for descriptions on the

icons and other options available on the Profiling Monitor tab.

You can also open snapshots in this view that you have saved from other sessions. Snapshots appear as a new profiling session.

Note: If you attach (**Attach - J Optimizer Agent** data collection option) to a process running outside the Workbench, it does not create a new process and the PID displayed corresponds to the remote application.

2. The Navigator tab

Use the Navigator tab to locate and drill down into your JBuilder and J Optimizer projects. View class and source code information for each project.

3. The Profile Configuration Wizard icon

Select the Profile Configuration icon to open the **Profile Configuration Wizard**. You use the profile configuration to specify a Java program to profile, the J Optimizer profiling tool with which to profile it, and to start the profiler. You also use a profile configuration to modify the settings for each tool to tailor profiling results. For example, you can configure the Request Analyzer to profile individual Java components, or all of a program's Java components. In addition, all of the profilers except the Thread Debugger allow you include or exclude specific classes during the profiling process. For more information, see the "About Using the Profile Configuration Wizard" Help page.

4. J Optimizer Data View tabs

The J Optimizer UI displays the data collected by each tool on separate tabs called "Views." A number of default views for each tool have been created and appear in the main section of the J Optimizer perspective. J Optimizer also provides a number of additional views to choose from. For a description of the views associated with each tool, click here.

5. J Optimizer Perspective button

If you are in a JBuilder perspective, click this button to display the J Optimizer perspective, which includes the default views and any other views you select from the Show Views window.

6. The J Optimizer Output Console

The Output Console displays the results when you run an application or applet from the J Optimizer perspective.

7. J Optimizer Server Activity Display

The Server tab displays the output that results when you run an application server in the J Optimizer perspective.

Other UI Elements

Element	Location and Purpose
Red **X** on a view tab	On J Optimizer view tabs. Indicates that the J Optimizer view does not contain live data (the selected application is in a terminated state)
Pause button and menu option	Resides on the **Profiling Monitor** tab toolbar and right-click menu. This button is not unique to J Optimizer, but its behavior is. In J Optimizer this button/option pauses the actual VM. It does not just pause polling (where the VM is still running)
CPU Profiling recording icons and menu options	Resides on the **Profiling Monitor** tab toolbar and right-click menu. Applies to the Profiler tool. Starts, stops, and sets options for CPU recording.
Generate Snapshot button/option	Resides on the **Profiling Monitor** tab toolbar and right-click menu. Applies only to Profiler, Code Coverage, and Request Analyzer tools. Use to compare or share data.
J Optimizer Tool Selector icon	The J Optimizer Tool Selector is used to start and configure remote or offline profiling sessions by those working from a command line.

Related Topics

- Overview of J Optimizer
- About Using the J Optimizer Views

1.3. Using the Stand-Alone J Opti Profile Configuration Wizard

J Optimizer provides a number of different settings for each profiling tool to help you tailor the data each tool collects. You specify these settings in a **Profile Configuration**. You use the **Profile Configuration Wizard** to perform the following tasks:

- Configure the profiling tool you want to use.
- Start profiling with the tool. You can start profiling immediately after creating a profile configuration, or, after saving the configuration, at another time.

When profiling completes, you can view the results in the J Optimizer UI using one of the data views provided for each tool. You can reuse a profile configuration to profile the same Java program with a different tool by specifying the different tool.

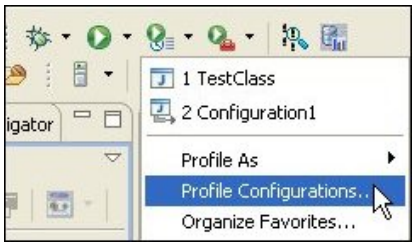
Because configuration settings vary by tool, separate **Help** topics have been created for each tool and are listed in the **Related Topics** section of this page. However, though the individual settings vary, the steps to open the wizard are the same for each tool and are provided below.

Note: If you have **Touchpoint J Optimizer** plugged into an Eclipse application, you can create a profile configuration from the Eclipse UI using the steps below. When profiling completes, you use Stand-alone J Optimizer to view profiling results.

Getting Started with the Profile Configuration Wizard

To open the Profile Configuration wizard:

1. Click on the Profile Configuration icon and select **Profile Configurations**, as shown below.

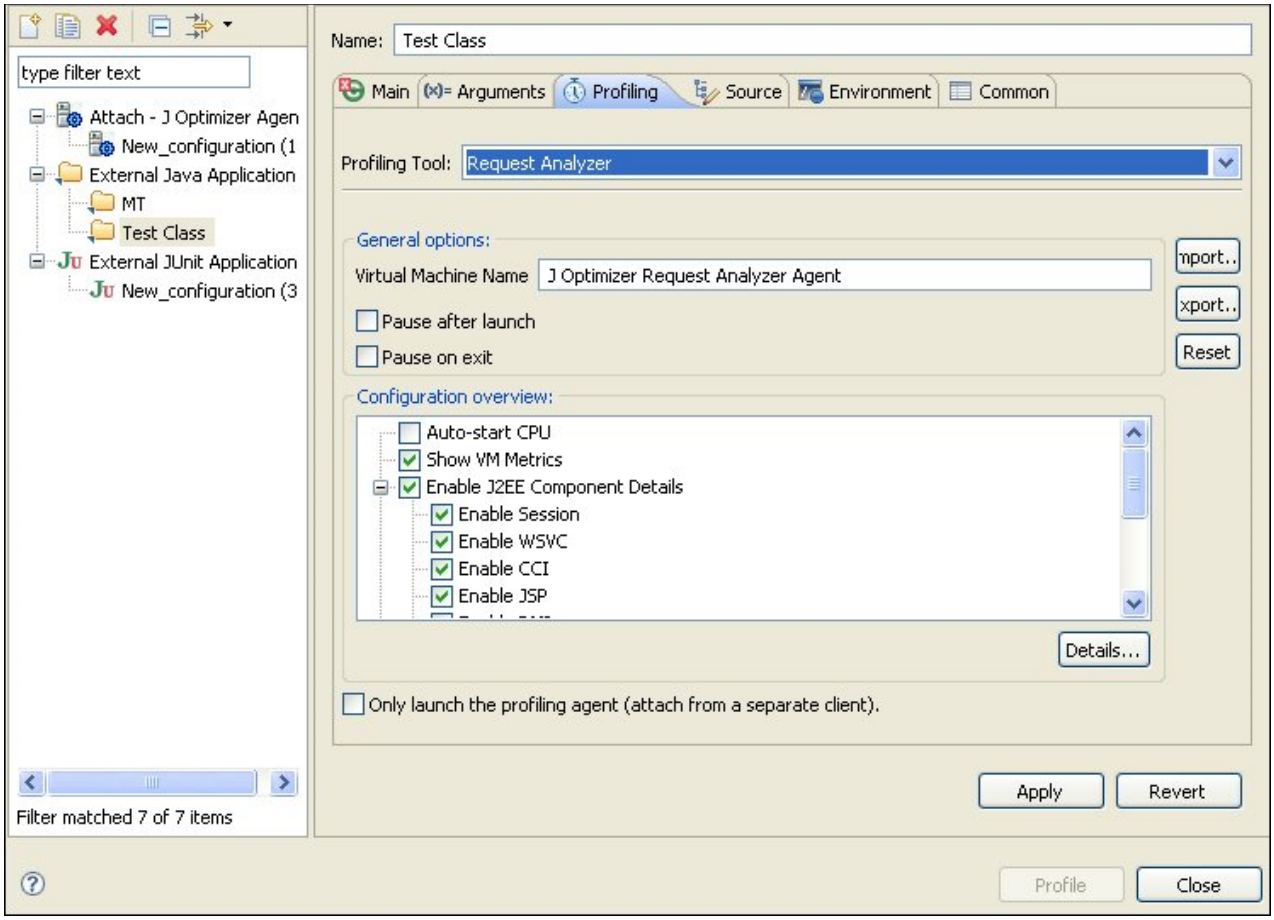


   The Profile Configuration wizard opens.

2. The **left pane** of the wizard displays the types of Java programs you can profile. Right-click on the type of program you want to profile, and select **New** to create a new configuration. You can also use an existing configuration. In the sample image below, the Test Class configuration, for an External Java Application, is selected.

   Note: To profile a server application, you use the Attach - J/Optimizer Agent option in the left pane. For instructions on how to profile a server application, click here.

3. This opens a series of tabs in the right pane. To select a J Optimizer profiling tool, click the **Profiling** tab.
4. Use the down-arrow in the **Profiling Tool** field to select a tool. In this example, the Request Analyzer is selected. Your tool selection determines which other fields appear on the Profiling tab.



5. Descriptions of the rest of the fields that appear on the **Profiling** tab after you select a tool are discussed in the **Help** pages listed in the **Related Topics** section below. These **Help** pages also explain the additional options that appear when you click the **Details** button on the Profiling tab. So, for example, for descriptions of the various CPU Profiler fields and options, click the Creating a CPU Profile Configuration link below. For descriptions of the Code Coverage fields and options, click Creating a Code Coverage Profile Configuration.

Related Topics

Creating a Memory Profile Configuration
Creating a CPU Profile Configuration
Creating a Code Coverage Profile Configuration
Creating a Thread Debugger Profile Configuration
Creating a Request Analyzer Profile Configuration
Creating an Application Server Profile Configuration

1.4. What's New in J Optimizer 2009

Since the previous release, following are new features in this release:

- **Support for** profiling on Solaris **machines**. J Optimizer now provides commands to support profiling on Solaris machines.
- **Branch support for** Code Coverage. The J Optimizer Code Coverage tool now provides automatic branch coverage. Branch coverage, sometimes known as "block" coverage, checks the testing status of each branch in a branching construct (such as an "if" or "case" statement).
- **Stand-alone J Optimizer**. Stand-alone J Optimizer provides full profiling capability without requiring integration into an IDE or a larger development environment. Stand-alone J Optimizer also includes the Touchpoint J Optimizer executable. **Touchpoint J Optimizer** is an integration plug-in that enables developers to use the J Optimizer profiling tools from within their own Eclipse-based development environment or IDE. Touchpoint J Optimizer does not have a graphical user interface (GUI) and is not sold separately. Finally, Stand-alone J Optimizer includes the Source Code Audit and Metrics tool.
- **New structure and content for J Optimizer online Help**. The J Optimizer Help system has been redesigned to provide more detailed instructions and easier access to the J Optimizer tools and options they want to use. Navigate through the J Optimizer Help Table of Contents (TOC) to locate information by tool and function. Separate Help topics for Touchpoint J Optimizer are available from the **Help>Help Contents** menu on your Eclipse application after you run the Touchpoint J Optimizer executable. The Source Code Audit and Metrics tool help is available from the **Help>Help Contents** menu in Stand-alone J Optimizer

For more information about the stand-alone and Touchpoint versions of J Optimizer, please contact Embarcadero Technologies.

Related Topics

- Overview of J Optimizer
- How to Get Started with J Optimizer

1.5. Supported Technologies, Servers, Platforms, and host IDEs

J Optimizer supports the following technologies, servers, platforms, and host IDEs that are specified in the following sections. J Optimizer support is identified by version: stand-alone, the JBuilder and Eclipse plugin versions.

Supported Technologies

With J Optimizer, you can profile applications that conform to many Java standards, including:

Supported Technology	Available for J Optimizer for JBuilder	Available for Stand-alone J Optimizer	Available for Touch Optimizer
J2SE 1.3, J2SE 1.4, J2SE 5.0 (1.5) and J2SE 6.0 (1.6)	X	X	X
J2EE 1.4, Java EE 5	X	X	X

Supported Servers

J Optimizer supports integration with most of the leading commercial and open source Java application servers, including:

Supported Servers	Available for J Optimizer for JBuilder	Available for Stand-alone J Optimizer	Available for Stand J Optimizer Ag
Apache Geronimo 1.1.1 and 2.0	X	X	X
Apache Tomcat 5.0, 5.5 and 6.0	X	X	X
BEA WebLogic Application Server 9.2, and 10.1	X	X	X
IBM WebSphere 6.1 (with EJB 3 feature pack)	X	X	X
JBoss 4.0.5, 4.2, and 5.0	X	X	X
JBoss 3.2.6	X	X	-
Oracle Application Server 10.1.3.3	-	-	X
Sun Glassfish 2.0	X	X	X
Sun Glassfish 1.1	X	X	-
Jetty 6.1	-	-	X

Supported Platforms

You can use J Optimizer on machines using the following operating platforms:

Supported Platforms	Available for J Optimizer for JBuilder	Available for Stand-alone J Optimizer	Available for Touchpoint J Optimizer	Available f Stand-alone Optimizer Ag
Microsoft Windows XP (SP3)	X	X	X	X
Microsoft Windows Vista (SP1)	X	X	X	X
Mac OS X (10.5)	X	X	X	X
Red Hat Enterprise Linux 5	X	X	X	X
Solaris 10 SPARC	-	-	-	X

Supported host IDEs

You can install Touchpoint J Optimizer on the following IDEs:

- Eclipse 3.3.2, and 3.4.1 Java and JEE editions
- MyEclipse 6.6, and 7.0
- IBM RAD 7.5

2. Overview of Memory Profiling

In the J Optimizer UI, the tool called **"Profiler"** can perform both Memory and CPU profiling (though not simultaneously). This **Help** page provides an overview of memory profiling. For an introduction to CPU profiling, click here.

The **Memory Profiler** can assist you in minimizing temporary object allocations and in detecting memory leaks. Temporary object allocation can slow down your application. Memory leaks are often caused by objects that continue to be referenced as the processes continue, so they are still using resources.

You can use the **Memory Profiler** to display all allocated instances in real time, and to see precisely which method is responsible for object allocations. The Memory Profiler also allows you to browse incoming and outgoing object references. Filters allow you to focus on relevant classes. Once you determine which class is causing a problem, you can immediately view the source code. Garbage collection controls are also provided. If desired, you can use API calls to invoke the Memory Profiler from within the program you are testing.

In short, you can use the Memory Profiler to:

- Identify memory leaks by examining the memory heap and comparing different heap states.
- Minimize temporary object allocations. Excessive temporary object allocations can cause the garbage collector to run every few seconds. When running, the garbage collector slows down your Java program.
- Minimize the number of instances required for a given operation, and therefore decrease the memory your program requires.
- Reduce memory waste by making sure every object is garbage collected.

You can specify and launch the Memory Profiler from within a memory profile configuration.

Note: The Memory Profiler reports memory use in bytes (B), kilobytes (KB), and megabytes (MB).

Heap Memory

When you start your profiling session, the Memory Profiler opens in the Heap view. This view displays all classes and the number of instances currently allocated. From this view you can mark the current instance count to see the instances allocated by a specific action in your program. After you have identified a class with an excessive number of instances, you can view the allocation backtrack to identify the code or the part of the program that is responsible for these allocations.

Garbage Collection

The Memory Profiler includes controls for garbage collection. From the Memory Profiler Heap view, you can study object allocations without having the garbage collector removing instances. In this case, the garbage collector is not really disabled; however, the Heap view shows you what would happen if the garbage collector was not running. You can also force the garbage collector to run immediately.

The Memory Profiler can also display:

- The amount of memory that would be garbage collected if the selected reference is deleted.
- References that are garbage collected only when memory is low.
- References that are garbage collected even if memory is abundant.
- References waiting in the finalizer queue to be garbage collected.

Memory Leaks

A memory leak prevents your program from reclaiming memory in the heap after it has finished using it. Memory leaks are often caused by objects that continue to be referenced as processes continue. These objects are still using resources. With the Memory Leak Detector, you can locate an object with numerous references and trace those references to locate the source of a memory leak.

About the Application Quality Analyzer

The **Application Quality Analyzer (AQA)** detects errors and warnings in the code being profiled. Types of errors reported include abnormal container growth, inefficient StringBuffer use, abnormal garbage collection duration, exception locations, abnormal finalizer queue length, and unclosed FileDescriptor objects. You can access AQA settings while creating or editing a CPU or Memory profile configuration.

Viewing Memory and CPU Metrics

The **Virtual Machine (VM) Metrics** analyzer returns class count, thread use, and heap size informance that helps you determine if a performance problem is related to CPU, memory, or both. You can instruct J Optimizer to collect VM Metrics when you run the **Memory or CPU Profiler**, or the **Request Analyzer**. For more information about collecting VM metrics, click here.

Related Topics

- About CPU Profiling
- Analyzing Application Quality
- About Collecting VM Metrics
- About Viewing Profile Results

2.1. Creating a Memory Profile Configuration

In the J Optimizer UI, the tool called "**Profiler**" can perform **Memory and CPU profiling** (though not simultaneously). This **Help** page explains how to use **Stand-alone J Optimizer** to create a profile configuration that profiles memory data. For instructions on creating a CPU profile configuration, see "Creating a CPU Profilie Configuration."

To create a new memory profile configuration:

1. Open the **Profile Configuration wizard**.
2. In the left pane, choose the Java program you want to profile.
3. In the right pane, click the **Profiling** tab.
4. Use the down-arrow in the **Profiling Tool** field to select the **Profiler** tool.
5. In the **General options** section, you can do the following:

Field name	Description
Virtual Machine Name	Change this name if desired. By default, J Optimizer creates a VM name based on the selected profiling tool.
Pause after launch	This option pauses the application before it executes the main method, thereby providing time to study the program's launch phase.
Pause on exit	When profiling a fast-running application, select this field to give the profiling tool time to complete data collection after the application stops running.
Enable audit API	Select this option to allow control of profiling directly from your source code.

6. In the Configuration Overview section, ensure that **Memory Profiling** is selected, then click the **Details** button.This generates the Configuration Details window.
7. On **Configuration Details,** click **Memory** in the left pane to display **Memory** settings in the right pane.
8. In the right pane, change the following memory profiling settings as needed. Then click **OK** to save your changes and close the Configuration Details window.

Field name	Description
Enable Memory Profiling	Enables the memory profiler to run.
Report Collected Objects	By default, the memory profiler reports on the number of collected objects. Deselect to disable this count.
Disable Garbage Collector	Select to disable the garbage collector while the profiler is running.
Show Allocations Since Mark	It can be useful to set a mark so that the Diff column shows all zeros. This makes it easier to isolate the objects that are generated and of interest to you.
Precision	"Use Method Precison" is the default. Select "Use Line Precison" to display the actual line number in your source code that is using the CPU and generating the objects.
Object Instances/Sizes	"Show Object Instances" is the default, and reports on the number of objects currently allocated. Select "Show Object Size" to report on the size (in bytes) of each object currently allocated.

Report Freed/Live Objects	"Report Only Live" is the default, and reports only objects that are that are still in the heap and not garbage collected. Select "Report Only Freed" to display only objects that are garbage-collection ready. "Report Live and Freed" shows objects that are still in the heap, not garbage-collected, and are garbage-collection ready.

9. This returns you to the Profiling tab, on which you can make the following additional selections:

Field name	Description
Auto-start CPU	This starts the CPU profile recording when you start the Profiler.
Quality Analyzer	By default, the Profiler collects Quality Analyzer data during profiling. Deselect this field if you don't want to receive this information. For more about Quality Analyzer information, click here.
Show VM Metrics	By default, the Profiler collects VM metrics during profiling. Deselect this field if you don't want to receive these metrics. For more about VM metrics, click here.
Use Filters	J Optimizer automatically enables filter creation for use with the Profiler. Deselect if you do not want to create filters. For more information about creating and using filters, click here.
Auto-Capture	Select this field to have J Optimizer generate snapshots during the profiling process. For more information about the auto-capture feature, click here.
Import	Click this button to import and edit a saved profiling configuration.
Export	Click this button to save this profile configuration and reuse it later as an imported profile configuration.
Only launch the profiling agent	Select this field if you plan to view profiling results on a separate client.

10. Click **Apply** to complete this profile configuration, or **Profile** to start profiling immediately.

What To Do Next

Use the J Optimizer viewing options to see and analyze the data collected by the profiling tool.

To **edit** a configuration, click on the Profile Configuration icon, then select the Profile Configurations option. This opens the wizard. In the left pane, click on the configuration, then on the tabs that contain the fields you want to change.

Related Topics

- Overview of Tool Configuration from the UI
- About the Memory and CPU Tool
- About Memory Profiling
- About Viewing Profile Results
- About the Profiling Quality Analyzer
- Configuring VM Metrics

2.2. Setting Filter Conditions

Create a J Optimizer **filter** to group classes into one or more categories, or to eliminate a class from profiling results. Filters can identify the memory and CPU usage that is associated with each class or group of classes in a filter. You can easily add and remove classes from a filter.

You can set filters for the Memory and CPU Profiler, and the Code Coverage and Request Analyzer tools. This **Help** topic provides instruction on how to create and edit filters from the J Optimizer user interface. For instructions on how to do so by modifying the J Optimizer configuration file, click here.

Setting and Editing Filters from the UI

You can set and edit filters from within a Memory, CPU, Code Coverage, and Request Analyzer profile configuration.

To set or edit filters:

1. In a profile configuration, open the **Configuration Details** dialog window.
2. In the left pane, click **Filters**. This makes filter-related fields available on the right side of the dialog.
3. Ensure that the **Enable Filtering** field is selected.
4. Click **Add** to create or edit a filter.
5. On the **Add Filter** dialog, take the following actions, then choose **OK**.

Field name	Description
Filter Element Name	Specify a filter name.
Classes to Filter	Choose Add to specify a filter pattern for each class.
Simplification	Many of the J Optimizer Views display a backtrace tree of methods. The Group option in the Simplification field allows you to "simplify" these backtraces merging those that match the filter pattern into one level, with the filter name as the displayed label. If you do not want to simply the backtrace tree, keep the default value selected, which is None.
Ignore CPU Usage	Select if you *do not* want the CPU profiler to run.
Ignore Memory Usage	Select if you *do not* want the memory profiler to run.
Combine filter patterns	Specifies what to do if a filter pattern matches a given element. The default is "Any of the listed classes," which means that if *any* of the filter patterns match the listed class, they will be filtered. "All of the listed classes" specifies that if *all* of the filter patterns match, they will be filtered.

6. When you have finished adding or editing filters, choose **OK**. This returns you to previous dialog window for the tool you are setting up.
7. Choose **Finish**. This returns you to the main Profile Configurations page.
8. Choose **Apply** to save your changes, or **Profile**, to start profiling immediately. Use the appropriate view tabs to view the data collected by the profiling tool. Results should reflect the filter patterns you set.

2.3. Analyzing Application Quality

The **Application Quality Analyzer** is a J Optimizer feature that you can run with the Memory and CPU Profiler. The Application Quality Analyzer detects errors and warnings in the code being profiled. The real-time validation routine checks your code for basic error conditions.

Specificially, the **Application Quality Analyzer** identifies the following types of errors and warnings:

- Abnormal container growth
- Inefficient StringBuffer use
- Abnormal garbage collection
- Exceptions thrown
- Abnormal finalizer queue length
- Unclosed file descriptor

Note: A different version of the Application Quality Analyzer can be launched from a Request Analyzer Profile Configuration. For more information, click here.

Setting Quality Analyzer Options

You can set Application Quality Analyzer settings when you create or edit Memory or CPU profile configuration. After you activate the Application Quality Analyzer settings and run the profile configuration, you can see the results in the **Application Quality Analyzer** data view.

To set Quality Analyzer options:

1. In the left pane of the **Configuration Details** window, select **Quality Analyzer**. The Quality Analyzer fields appear in the right pane.
2. In the right pane, change the following settings as needed. Then click OK to save your changes and return to the J Optimizer Profiler dialog.

Field name	Description
Enable Quality Analyzer	Select to enable Quality Analyzer data collection.
Finalizer Queue Length	Specify a maximum Finalizer Queue Length. An error is reported when the queue length is exceeded. This may indicate, for example, that the finalize () method is overused in your code. The default value is 1000.
Pause Delay	Specify a maximum Pause Delay, in milliseconds. An error is reported when J Optimizer detects excessive garbage collection activity. The default value is 100 miliseconds.
Allocation Count	Specify a maximum StringBuffer Allocation Count. An error is reported when the number of string buffer allocations per method invocations exceeds the maximum allocation count specified. The default value is 10.
Exception Classes	Indicate Exception Classes to exclude and/or include. Exceptions are reported when the name of the class throwing the exception matches the pattern specified.
Byte-based container growth	Specify the Minimum Size and Growth Count for byte-based container growth. An error is reported when the minimum size or growth count is exceeded. The default values are 1024 bytes and a count of 2.
Array-based container growth	Specify the Minimum Size and Growth Count for array-based container growth. An error is reported when the minimum size or growth count is exceeded. The default values are 100 and 2.
Hashtable-based container growth	Specify the Minimum Size and Growth Count for hashtable-based container growth. An error is reported when the minimum size or growth count is exceeded. The default values are 101 and 2.

3. On the J Optimizer Profiler dialog, click **Finish**.
4. On the main Profile Configuration Wizard page, click **Apply** save your changes, or **Profile** to start profiling immediately. You can see the results in the J Optimizer **Application Quality Analyzer** data view. To access this view, select **Window** from the main menu, then **Show Views**, followed by **Other...**. The Application Quality Analyzer view is listed near the top of the J Optimizer views.

2.4. Configuring VM Metrics

The **VM Metrics** analyzer is a J Optimizer feature that you can run with the Memory and CPU Profiler or the Request Analyzer. You can the VM Metrics analyzer to gather the following performance information on the program you want to profile:

- Number of loaded classes
- Heap size allocated to Java objects
- Number of active threads
- Garbage collector activity

For more detailed descriptions of the information collected by the VM analyzer , click here.

Setting VM Metrics Options

Use the instructions below when you are in the process of creating or editing a Memory, CPU, or Request Analyzer profile configuration. After you activate the Metrics settings and run the profile configuration, you can see the results in the J Optimizer **Metrics** data view.

To set VM Metrics options:

1. In the left pane of the **Configuration Details** window, select **Metrics**. The Metrics fields appear in the right pane, as shown below.
2. In the right pane, ensure that the **Show Metrics** field is selected.
3. In the **Java heap** settings section, indicate the following if you want to view heap metrics:

Field name	Description
Show java heap metrics	This field is selected by default
Update interval	Specify seconds, minutes or hours
Buffer Type	Specify time or size
Buffer Size	Specify seconds, minutes or hours

4. In the **Thread count** settings section, select the **Show thread count metrics** if you want to view thread metrics. Then set the thread interval and buffer fields as desired.
5. In the **Class count** settings section, select the **Show class count metrics** field to view class methods. Then set the class interval and buffer fields accordingly.
6. In the **Show garbage collection** settings section, select the **Show garbage collection metrics** field to view garbage collection metrics. Then set the garbage collection interval and buffer fields as desired.
7. When you are finished, click **OK**. This saves your settings returns you to the Profile Configuration Wizard from which you can **Finish** the configuration or change more settings. When you run this profile configuration, you can see the results in the J Optimizer **Metrics** data view.

Related Topics

About Collecting VM Metrics

2.5. About Unit Test Profiling

You can use the J Optimizer CPU Profiler and the Memory Profiler to gather, display, and compare performance information for **JUnit tests**. Performance data is gathered and automatically stored in a snapshot for each test. These snapshots include performance data provided by the CPU and Memory Profiler. JUnit test profiling is supported by the J Optimizer Profiler within the user interface and from the command line.

Viewing Unit Test Profiling Results

After each individual test finishes executing, the J Optimizer Profiler runs the garbage collector, marks the current instance count, and clears any errors captured by the Application Quality Analyzer.

If you start a testing session for unit tests from the command line, you must be attached to the J Optimizer Agent while the tests are running to access the **Unit Test View**. When you start a testing session within the user interface, the Unit Test view opens automatically, and reports the progress of the different unit tests as they run. Once all tests have been executed, you can view **snapshots** for individual tests, as explained in the procedure below.

The Unit Test view lists the tests that have run during a given testing session and reports the following information for each test:

- Test status, passed or failed
- Amount of time the test took to execute
- Number of objects allocated
- Number of objects garbage collected
- Amount of Memory allocated by object creation
- Amount of Memory freed

To display the JUnit Test View, select **Window>Show View>Other**. In the Show View dialog, under J Optimizer, select **Unit Test Snapshots** to insert this view into the J Optimizer perspective.

How to Profile JUnit Tests and View Results

To run the profiler for a unit test and view results:

1. Open the **Profile Configuration wizard.**
2. In the left pane, choose the **JUnit Test** you want to profile.
3. In the right pane, select the **Profiler** tool, then click **Edit Options** to view Profiler options.
4. On the next dialog, click **Details**.
5. On **Configuration Details**, in the left pane, click **Unit Test Snapshots.**

6. In the right pane, ensure that **Enable Snapshot-Generation for Unit Tests** is enabled.
7. In the Snapshot generation options section, you can take the following actions:
    - **Report Memory** instructs J Optimizer to run the memory profiler in addition to the CPU profiler. Deselect if you don't want J Optimizer to profile memory usage.
    - **Filter JUnit classes** instructs J Optimizer to collect and display only non-JUnit classes. Deselect to include JUnit class information.
8. Select a directory in which to store the snapshots.
9. Click **OK** to save your changes.
10. On the Profiler diaglog, click **Finish**.
11. On the main Profile Configurations wizard page, click **Apply** to save your changes, and/or **Profile** to start profiling results.
12. To view the JUnit test snapshots when profiling completes, right-click on the test process in the **Profiling Monitor** and select Optimizit, followed by the name of the snapshot (CPU activities or Memory activities). The snapshot appears in the right pane.

3. Overview of CPU Profiling

In the J Optimizer UI, the tool called **"Profiler"** can perform both Memory and CPU profiling (though not simultaneously). This Help page provides an overview of CPU profiling. For an overview of memory profiling, click here.

Use the **CPU Profiler** to identify which methods your program uses and help you understand what to change to improve performance. The CPU Profiler displays profiling results for each thread or thread group for pure CPU use or for elapsed time (pure CPU and inactive phases). You can start or stop profiling at any time, with millisecond or microsecond precision. The CPU Profiler has both **sampler-based** and **instrumentation-based** modes, and also contains filters to remove fast methods.

The CPU Profiler displays the methods used by hot spots. **Hot spots** are methods where the most time is spent. The time shown is the time the program spent in a method, no matter where the method was called from. You can determine if a single method acts as a bottleneck and can be optimized to speed up all the tested features. A backtrace tree showing how time was spent or how the CPU was used during the testing session can also be displayed, helping you understand what to change to improve performance.

You can select and launch the CPU profiler from within a CPU profile configuration and a Request Analyzer profile configuration.

Understanding CPU Profiler Output

The CPU Profiler displays a call stack trace for a thread or thread group that executed during the profiling session. By default, the CPU Profiler displays data for the thread responsible for the most CPU activity. You can select a thread or thread group from the drop-down thread menu at the top of the CPU Profiler view. Selecting a thread group shows how the time was spent for all threads and thread groups belonging to the thread group.

The call stack trace can be displayed as a backtrace tree, a graph, or a table. The stack trace information can help you track down the source code responsible for CPU utilization or bottlenecks.

Tip: Click **Reverse Display** to reverse the backtrace tree from the leaves to the root. This view can be useful when you need to focus on methods or lines of code rather than broad features in your test program.

The CPU Profiler view also includes a table that lists hot spots. Hot spots are methods where the most time is spent. The time shown is the time the program spent in a method, no matter where the method was called from. You can determine if a single method acts as a bottleneck and can be optimized to speed up all the tested features.

About the Two CPU Profilers

J Optimizer provides two types of CPU Profilers:

- Sampler
- Instrumentation

One type of profiler may work better for your code than the other. The sampler profiler is very good at testing a large amount of code for a long time. The instrumentation profiler is very good at precisely testing a small amount of code. The instrumentation profiler also shows if a method is slow or if it is called too often.

The type of profiler you are using determines what options you have to refine your CPU Profiler display.

Note: The default J Optimizer filters aggressively filter code. By default, no non-public methods will be reported, as they are all filtered. You may want to remove the filter conditions for these methods, depending on your needs.

Sampler Profiler

The sampler profiler interrupts all running threads periodically. The sampling period is specified by the user. Once all threads are interrupted, the CPU Profiler records what each thread is currently doing and whether each thread is currently using CPU. It then resumes all running threads.

The sampler profiler is recommended for the following testing situations:

- Testing a program for a very long time, for example, testing a server overnight. This is because sampler has very low overhead and excellent scalability.
- Testing a feature that requires a lot of different code, for example, the startup of a large UI-intensive application. Because sampler pauses all threads before recording any information, it does not distort performance data. Sampler can also detect performance bottlenecks within methods because it is not based on method invocations.

Note the following disadvantages of the sampler profiler:

- Lack of precision: the precision of the sampler profiler is not greater than its sampling period.
- Does not record the number of method invocations.

Instrumentation Profiler

The instrumentation profiler, which is recommended for precision testing, intercepts method invocations. Each time a method is called, the CPU Profiler records the fact that a method was called and gives the control back to the application. The profiler also intercepts when a method returns from executing and records the amount of time or CPU that was spent in the method.

The instrumentation profiler is recommended for the following testing situations:

- Testing anything that executes in less than a few hundred milliseconds, for example, a menu action. Instrumentation can measure precision in microseconds, and each time a method is invoked, it is recorded.
- Testing a system that has many threads executing many small requests, for example, a servlet. Instrumentation records the number of times each method is invoked.

Note the following disadvantages of the instrumentation profiler:

- Lack of scalability: this profiler records a lot of information.
- Information distortion: this profiler actually runs in the tested program threads. All method invocations are slower. Even if the profiler compensates, this can lead to distorted results.
- Large overhead: the tested application runs several times slower.

About the Application Quality Analyzer

The **Application Quality Analyzer (AQA)** detects errors and warnings in the code being profiled. Types of errors reported include abnormal container growth, inefficient StringBuffer use, abnormal garbage collection duration, exception locations, abnormal finalizer queue length, and unclosed FileDescriptor objects. You can access AQA settings while creating or editing a CPU or Memory profile configuration.

Viewing Memory and CPU Metrics

The **Virtual Machine (VM) Metrics** analyzer returns class count, thread use, and heap size informance that helps you determine if a performance problem is related to CPU, memory, or both. You can instruct J Optimizer to collect VM Metrics when you run the **Memory or CPU Profiler**, or the **Request Analyzer**. For more information about collecting VM metrics, click here.

Related Topics

About Memory Profiling

About Analyzing Application Quality

About Collecting VM Metrics

About Viewing Profile Results

3.1. Creating a CPU Profile Configuration

In J Optimizer UI, the tool called **"Profiler"** can perform **Memory and CPU profiling** (though not simultaneously). This **Help** page explains how to use **Standalone J Optimizer** to create a profile configuration that profiles CPU data. For instructions on creating a memory profile configuration, see "Creating a Memory Profile Configuration."

The CPU profiler offers two types of CPU profiling: the **Sampler** profiler is very effective at testing a large amount of code for a long time. The **Instrumentation** profiler is very effective at precisely testing a small amount of code. The instrumentation profiler also shows if a method is slow or if it is called too often. You specify the type of CPU profiling you want to use when creating the CPU profile configuration.

To create a new memory profile configuration:

1. Open the **Profile Configuration wizard**.
2. In the left pane, choose the Java program you want to profile.
3. In the right pane, click the **Profiling** tab.
4. Use the down-arrow in the Profiling Tool to select the **Profiler** tool.
5. In the **General options** section, you can do the following:

Field name	Description
Virtual Machine Name	Change this name if desired. By default, J Optimizer creates a VM name based on the selected profiling tool.
Pause after launch	This option pauses the application before it executes the main method, thereby providing time to study the program's launch phase.
Pause on exit	When profiling a fast-running application, select this field to give the profiling tool time to complete data collection after the application stops running.
Enable audit API	Select this option to use J Optimizer API code in your tested application to control CPU Profiler and Memory Profiler operation.

6. In the Configuration Overview section, ensure that **Auto-start CPU** is selected, then click **Details**. This generates the Configuration Details window.
7. On **Configuration Details,** click **CPU** in the left pane to display **CPU** settings in the right pane.
8. In the right pane, change the following CPU profiling settings as needed, then click **OK**. This saves your changes and closes the Configuration Details window.

Field name	Description
Auto-start CPU profiler	Select to use the CPU profiler.
Instrumentation	Select to enable Instrumentation-type CPU profiling; this is the default setting for the CPU Profiler. For details on how instrumentation profiling works, click here.
Instrumentation options	--In the **Precision** field, specify microseconds if you do not want data collected in milliseconds, the default. --Select **Enable instrumentation filter** to filter fast methods from the output. --Select **Enable Root Filtering** to add a filter pattern to tailor data collection.
Sampler	Select to enable Sampler-type profiling. For details on how sampler profiling works, click here.
Sampler options	--The **Precision** attribute can be set to method or line to control the precision of the CPU Profiler output. Set this attribute to method to aggregate data by method. --The **Period** attribute specifies the sampling period in milliseconds, which controls the granularity of the output. Use a small value for a short test session and larger value for a long test session. This value is typically in the range of 1 to 100.
Record only CPU usage	Select to record only CPU usage.
Recording options	You can set the CPU profiler to collect data at timed intervals, or you can start and stop it manually at any time during your profiling session.

9. This returns you to the Profiling tab, on which you can make the following additional selections:

Field name	Description
Auto-start CPU	Select to start the CPU profiler recording when you start the Profiler.
Quality Analyzer	By default, the Profiler collects Quality Analyzer data during profiling. Deselect this field if you don't want to receive this information. For more about Quality Analyzer information, click here.

Show VM Metrics	By default, the Profiler collects VM metrics during profiling. Deselect this field if you don't want to receive these metrics. For more about VM metrics, click here.
Memory Profiling	If you select this field, the memory Profiler will start reporting data after the CPU Profiler recording stops.
Use Filters	J Optimizer automatically enables filter creation for use with the Profiler. Deselect if you do not want to create filters. For more information about creating and using filters, click here.
Auto-Capture	Select this field to have J Optimizer generate snapshots during the profiling process.
Import	Click this button to import and edit a saved profiling configuration.
Export	Click this button to save this profile configuration and reuse it later as an imported profile configuration.
Only launch the profiling agent	Select this field if you plan to view profiling results on a separate client.

10. Click **Apply** to complete this profile configuration, or **Profile** to start profiling immediately.

What Happens Next

Use the J Optimizer viewing options to see and analyze the data collected by the profiling tool.

To **edit** a configuration, click on the Profile Configuration icon, then select the Profile Configurations option. This opens the wizard. In the left pane, click on the configuration, then on the tabs that contain the fields you want to change.

Related Topics

- Overview of Tool Configuration from the UI
- About Profiling CPU Usage
- About Viewing Profile Results
- About the Profiling Quality Analyzer
- Viewing VM Metrics

4. Overview of Code Coverage

J Optimizer Code Coverage allows you to determine the exact lines of source code that your application is executing. In real time, you can see how frequently each class, method, branch, and line of code is executed. You can use J Optimizer Code Coverage to test applications, applets, JavaBeans, Enterprise JavaBeans (EJBs), JavaServer Pages (JSPs), and virtually any other Java code. You specify Code Coverage settings in a Code Coverage Profile Configuration.

J Optimizer Code Coverage is designed to help you identify and analyze the following information about your code:

- **Dead code**: Every line of code in a program should be functional. Dead code, occurring when code is dormant or cannot be accessed, may reveal logic errors. Dormant code makes an application longer and more difficult to understand. You can use Code Coverage to locate dormant code. You can also use Code Coverage to identify how much of your code has been used by highlighting all tested code. Understanding exactly which classes, methods, and lines of code have not been used allows you to modify your test plan to cover all areas of the code.
- **Frequently used code**: Identify the classes and methods in your code that are used most frequently. Performance optimization on frequently used code can have a substantial impact on overall performance. To find frequently used code, open the Method Coverage windows for the class you want to examine. The Invocation # column shows the number of times the selected method was invoked, representing how frequently (or infrequently)it is used.
- **Unloaded classes**: Unloaded classes are classes that exist in your class path but are not called. You can use the J Optimizer Code Coverage report to identify the number of classes never loaded by the virtual machine. The report lists unloaded classes by name. This report is available in HTML, CSV, or XML formats.

Code Coverage Data Views

J Optimizer can display the data collected by Code Coverage in two ways:

- **Class coverage view**: Use the Class Coverage view to identify a class you want to investigate. Once you have identified a class to investigate, you can use the Method Coverage view for more detailed code usage information.
- **Method coverage view**: To view code usage detail, select a class and double-click it. Method Coverage will display the source code for your method as well as statistical information about the number of times your method was called while the target application was running.

The **Coverage** column in the Class Coverage and Method Coverage windows display the percentage of a class or method that can be executed.

You can also save test information as a snapshot, which can be viewed later for further analysis. This is useful, for example, when you are testing several different classes in one test pass. Snapshots are generated from within the user interface or with a specific command option when running the test program.

Related Topics

Generating a Code Coverage Report

About Viewing Profile Results

4.1. Creating a Code Coverage Profile Configuration

The instructions on this **Help** page explain how to configure and launch the **Code Coverage** tool using **Stand-alone J Optimizer**.

To create a new code coverage profile configuration:

1. Open the **Profile Configuration wizard.**
2. In the left pane, choose the Java program you want to profile.
3. In the right pane, click the **Profiling** tab.
4. Use the down-arrow in the **Profiling Tool** field to select the **Code Coverage** tool.
5. In the **General options** section, you can do the following:

Field name	Description
Virtual Machine Name	Change this name if desired. By default, J Optimizer creates a VM name based on the selected profiling tool.
Pause after launch	This option pauses the application before it executes the main method, thereby providing time to study the program's launch phase.
Pause on exit	When profiling a fast-running application, select this field to give the profiling tool time to complete data collection after the application stops running.
Generate snapshot when exiting	Select to generate a snapshot file of code coverage profiling data before the profiler finishes.

6. In the **Configuration overview** section, deselect **Use filters** if you don't want to use filters for this profiling session. Click the **Details** button if you do want to set up data collection filters.
7. Click **Import** to import and edit a profile configuration from a saved profiling session.
8. Click **Export** to save this profile configuration and reuse it later as an imported configuration.
9. Select the checkbox in the **Only launch the profile agent** field if you plan to view profiling results on a separate client.
10. Click **Apply** to complete this profile configuration, or **Profile** to start profiling immediately.

What Happens Next

Use the J Optimizer viewing options to see and analyze the data collected by the profiling tool.

To **edit** a configuration, click on the Profile Configuration icon, then select the Profile Configurations option. This opens the wizard. In the left pane, click on the configuration, then on the tabs that contain the fields you want to change.

Related Topics

About the Code Coverage Tool

Overview of Tool Configuration from the UI

About Viewing Profile Results

How to Generate a Code Coverage Report

4.3. Generating a Code Coverage Report

At the command line, you can generate a report from a testing snapshot using the ReportGenerator class. A coverage report presents the test information for all covered classes. You can customize your report with ReportGenerator options.

Report Generator Syntax

After you have modified you environment variables, invoke the J Optimizer Code Coverage ReportGenerator class:

```
java intuitive.optit.coverage.ReportGenerator
 [-methodInfo] [-showDiff] [-showSignatures] [-reportType FileType]
 [-verbose] [-showSource] [-sourcePath SourcePath] snapshot report
```

Should you need help setting the appropriate environment variables, refer to the following code snippet as a working example:

```
SET OPTI_HOME=C:\JOptimizer
SET JAVA_HOME=C:\jdk1.5.0
SET CLASSPATH=%OPTI_HOME%\joptimizer-agent\lib\optit.jar;%CLASSPATH%
SET PATH=%OPTI_HOME%\joptimizer-agent\bin;%PATH%
%JAVA_HOME%\bin\java -Djava.library.path=%OPTI_HOME%\joptimizer-agent\bin intuitive.optit.coverage.Repo
```

The following table describes the ReportGenerator command options:

Option	Description
`-methodInfo`	Specifies to include the method test information. For each tested class, the report includes a table with the test results for each method of the class.
`-showDiff`	Specifies to include the test information since the mark.
`-showSignatures`	Displays the full name of methods, including their signatures.
`-reportType`	Specifies the type of report generated. Follow this option with either HTML or ASCII. The default value is HTML.
`-verbose`	Prints information about the status of the report generation.
`-showSource`	Specifies to include the available source code and line coverage for the tested classes. This option is only effective with the -methodInfo option.
`-sourcePath`	Specifies the location for the source code when the -showSource option is used. Follow this option with the path to the source files of the classes tested.

Command Examples

The following Windows command generates the report stressTest.html from the snapshot stressTest.snp, including the method coverage information.

```
java intuitive.optit.coverage.ReportGenerator
 -methodInfo c:\Test\stressTest.snp c:\Test\stressTest.html
```

On Linux, the same command would look like the following:

```
java intuitive.optit.coverage.ReportGenerator
 -methodInfo /home/user/Test/stressTest.snp /home/user/Test/stressTest.html
```

The following command generates the report EJBs_coverage.html from the snapshot test5.snp, including the method information. The methods display with their full signature. The source code with line coverage information is included. The source file for the tested classes is located under the directory c:\EJB_src.

```
java intuitive.optit.coverage.ReportGenerator
 -methodInfo -showSignatures -showSource
 -sourcePath c:\EJB_src test5.snp EJBs_coverage.html
```

On Linux, the same command would look like the following:

```
java intuitive.optit.coverage.ReportGenerator
 -methodInfo -showSignatures -showSource
 -sourcePath /home/user/EJB_src test5.snp /home/user/EJBs_coverage.html
```

5. Overview of Thread Debugger

The J Optimizer Thread Debugger displays real-time threading information for Java applications, applets, and JavaBean components. You can see how your program uses computer resources, as well as identify thread contentions, thread starvation, excessive locking, and deadlocks. The J Optimizer Thread Debugger provides automatic thread and monitor usage reports that help developers prevent deadlocks and other thread issues before they occur. You specify Thread Debugger options in a Thread Debugger Profile Configuration.

About the Thread Debugger Data Views

The J Optimizer Thread Debugger provides three main views for displaying data collected by the Agent:

- **Thread Display**: Use this display to view thread activity.
- **Monitor Display**: Use this view to analyze thread deadlocks.
- **Monitor Usage Analyzer**: This view can help predict deadlocks before they occur.

By default, the J Optimizer Thread Debugger starts in the Thread Display view. If you select a thread in the Thread Display view, you can access additional views of detailed activity information for individual threads.

Using the Realtime Thread Display

The Real Time Thread Display provides a real-time display of all threads used by the test program as well as their progress in a monitor usage graph. For each thread, you can see how many monitors are owned and how many monitors were entered. For threads waiting on monitors, you can determine how long each thread has been blocked and where threads are waiting.

The Real Time Thread Display also:

- Displays where blocked threads are waiting for monitors (includes monitors and threads involved in the contention).
- Displays how long each thread waiting for Input/Output operations has been waiting and where it is waiting.
- Automatically highlights lines of code relevant to thread activity.
- Allows specific time-range queries for details such as code execution location.
- Displays locking situations in real time in an easy-to-understand graph.
- Allows graph links to be selected. These links show where a thread entered a monitor or where a thread is blocking for a monitor.

From an individual thread in the Threads Display view, you can navigate to the following views for tracing specific activity information for a given thread back to the source code:

- Contention view
- Waiting and I/O-Waiting views
- Monitor Enter view

These views are accessed from J Optimizer Thread Debugger toolbar.

Contention View

The Contention view provides all the data necessary to understand why a contention occurs for a monitor. To display the Contention view, select a thread in the Thread Display view that has been blocking for some monitors and then click the Contention View button.

Waiting and I/O-Waiting Views

The Waiting and I/O-Waiting Views describe why a thread is not making progress. Both views apply to the thread selected in the Thread Display view.

- The Waiting view displays where a thread is waiting for a monitor.
- The I/O-Waiting view displays where a thread is blocked on an Input/Output (I/O) operation. J Optimizer Thread Debugger assumes an I/O operation is taking place if the thread is not making any progress in native code for a few milliseconds or more.

Monitor Enter View

The Monitor Enter View describes where a thread enters and locks monitors. Use this information to understand and correct excessive locking. The Monitor Enter View is used to study the optimum number of monitors to perform an operation. Although virtual machines are

becoming faster at unlocking unused monitors, entering unnecessary monitors may degrade performance by creating unnecessary contentions.

Using the Realtime Monitor Display

The Real Time Monitor Display provides a real-time display of thread and monitor relationships, and allows each relationship to be individually selected and viewed. It also provides stack traces of blocked methods and those acquiring monitors.

Using the Monitor Usage Analyzer

The Monitor Usage Analyzer records monitor usage patterns while the Thread Debugger is running. It automatically reports warnings and errors about runtime situations that can lead to deadlocks. Additionally, it reports errors when:

- Different threads use multiple monitors in a different sequence.
- Threads enter a monitor and wait for another monitor.
- Threads enter a monitor and wait on an I/O operation

Related Topics

About Identifying Thread Problems

About Viewing Profile Results

5.1. Creating a Thread Debugger Profile Configuration

The instructions on this **Help** page explains how to configure and launch the **Thread Debugger** tool using **Stand-alone J Optimizer**.

To create a new thread debugger profile configuration:

1. Open the **Profile Configuration wizard**.
2. In the left pane, choose the Java program you want to profile.
3. In the right pane, click the **Profiling** tab.
4. Use the down-arrow in the **Profling Tool** field to select the **Thread Debugger** tool.
5. In the **General options** section, you can take the following actions:

Field name	Description
Virtual Machine Name	Change this name if desired. By default, J Optimizer creates a VM name based on the selected profiling tool.
Pause after launch	This option pauses the application before it executes the main method, thereby providing time to study the program's launch phase.
Pause on exit	When profiling a fast-running application, select this field to give the profiling tool time to complete data collection after the application stops running.

6. In the **Configuration overview** section, select **Auto-start Monitor Analyzer** to record monitor usage while the thread debugger is running.
7. If you selected the Auto-start box in step 6, click the **Details** button to specify how you want the Analyzer to record time.
   The Configuration Details dialog for the Monitor Analyzer appears. In the Monitor Analyzer section, change the following settings as needed, then click **OK**.

Field name	Description
Manual recording	This default setting allows you to start and stop the Monitor Analyzer recorder as needed.
Timed recording	Select to specify how often, in seconds or miliseconds, you want the Monitor Analyzer to record monitor usage.

8. Back on the **Profiling** tab, select the checkbox in the **Only launch the profile agent** field if you plan to view profiling results on a separate client.
9. Click **Import** to import and edit a profile configuration from a saved profiling session.
10. Click **Export** to save this profile configuration and reuse it later as an imported configuration.

11. Back on the Monitor tab, click **Apply** to complete this profile configuration, and **Profile** to start profiling immediately.

What Happens Next

Use the J Optimizer viewing options to see and analyze the data collected by the profiling tool.

To **edit** a configuration, click on the Profile Configuration icon, then select the Profile Configurations option. This opens the wizard. In the left pane, click on the configuration, then on the tabs that contain the fields you want to change.

Related Topics

About the Thread Debugger Tool
About Identifying Thread Problems
Overview of Tool Configuration from the UI
About Viewing Profile Results

5.2. Identifying Thread Problems

Use J Optimizer Thread Debugger to identify the following threading problems:

- Thread contention
- Thread starvation
- Excessive thread locking
- Deadlocks

This information can be used to improve the performance and reliability of your Java applications. You specify thread debugger options in a Thread Debugger Profile Configuration.

Thread Contention

A contention is a situation where two or more threads are trying to enter the same monitor. Multi-threaded Java applications can experience severe performance bottlenecks if many threads attempt to acquire the same monitors at the same time. The J Optimizer Thread Debugger Contention view provides all the data necessary to understand why a thread contention occurs for a monitor.

Many contentions are caused by either the locking granularity or thread traffic.

Locking Granularity Too High

Keeping critical sections as small as possible allows for maximum thread concurrency. When a critical section is too long, the locking granularity is often too high. The critical section is the section of code executed while the monitor is held. The critical section can be either too slow, or simply too large.

When the critical section takes too long, Thread Debugger shows that a few threads experience long contentions. Also, threads usually enter monitors very high in the call graph.

Often the best resolution is to use a monitor lower in the call stack. Consider removing any monitor that has a large scope and use one or more monitors to protect data structures only while performing thread-unsafe computations.

Too Many Threads Using the Same Monitors

The issue may not be locking granularity. There may be too many threads competing for the same monitors, causing major contentions. In such situations, the Thread Debugger shows all the threads blocking for the same monitors.

Avoid this situation by duplicating the resource each thread is trying to acquire and implementing a deterministic way for threads to always go to the same resource.

For example, assume a program is experiencing some serious contentions while acquiring a monitor that is protecting a cache. All threads in this program need the cache, so if cache operations are too small, contentions will exist. It is possible to resolve this situation by creating N caches and making sure that the code accessing the cache always goes to cache i, where i is deterministically derived from the thread hash code (for example, thread.hashCode() % N).

Thread Starvation

Thread starvation occurs when one or more threads tie up CPU resources, preventing other threads from executing. This can be caused by poor scheduling or setting thread priorities inappropriately. The Thread Display in J Optimizer Thread Debugger provides a real-time view of threads currently running within the virtual machine, making it easy to witness thread behavior and to understand if threads are running

normally. This display can help you understand and resolve thread starvation for a given resource.

Excessive Thread Locking

Excessive thread locking occurs when a program enters many monitors without having a real need for the provided synchronization. This can occur, for example, when too many methods are synchronized, or when several monitors are used to protect the same critical section. The Monitor Enter view in J Optimizer Thread Debugger describes where a thread enters and holds monitors. You can use this information to understand and correct excessive locking.

Deadlocks

One of the most challenging aspects of multi-threaded programs is assuring that threads never deadlock. A deadlock occurs when two or more threads cannot make progress because they require the same unavailable monitor. The monitor is unavailable because the thread owning the monitor lock is itself blocking on another unavailable monitor, waiting for a monitor or an I/O operation.

The Monitor Display view helps you understand the cause of deadlocks that occur during a testing session by providing a real-time graph showing the relationship of threads and monitors. If the deadlocking behavior is intermittent, use the Monitor Usage Analyzer view, which predicts deadlocks before they occur.

6. Overview of Request Analyzer

All J Optimizer tools can profile JEE applications, but only the **J Optimizer Request Analyzer** can capture JEE-related events. The Request Analyzer is designed specifically for JEE developers to identify and correct application programing problems during the development phase of an application. When you integrate the Request Analyzer with your application server, the Request Analyzer captures and records component and container activity while the application is in use. You specify Request Analyzer options and settings in a Request Analyzer profile configuration.

You can use the J Optimizer Request Analyzer to:

- **Gather key performance metrics** for JDBC, JSP, JMS, and other JEE elements.
- **Error tracking**, including source code locations for errors and suggested solutions.
- **Continuous JEE profiling**, concurrent with VM profiling, to track performance issues as they occur.
- **CPU profiling** and dynamic high-level VM information.

The Request Analyzer also captures remote method invocations, and can report when your application is accessing Java objects running in another VM using Java Remote Method Invocation (RMI). Likewise, if the attached VM receives RMI calls, the Request Analyzer reports the calling method and the method called.

Request Analyzer results can be displayed in a variety of ways to help you isolate and analyze application code that may cause performance, scalability, or reliability problems. Details are provided in the following section.

Using the Request Analyzer Views

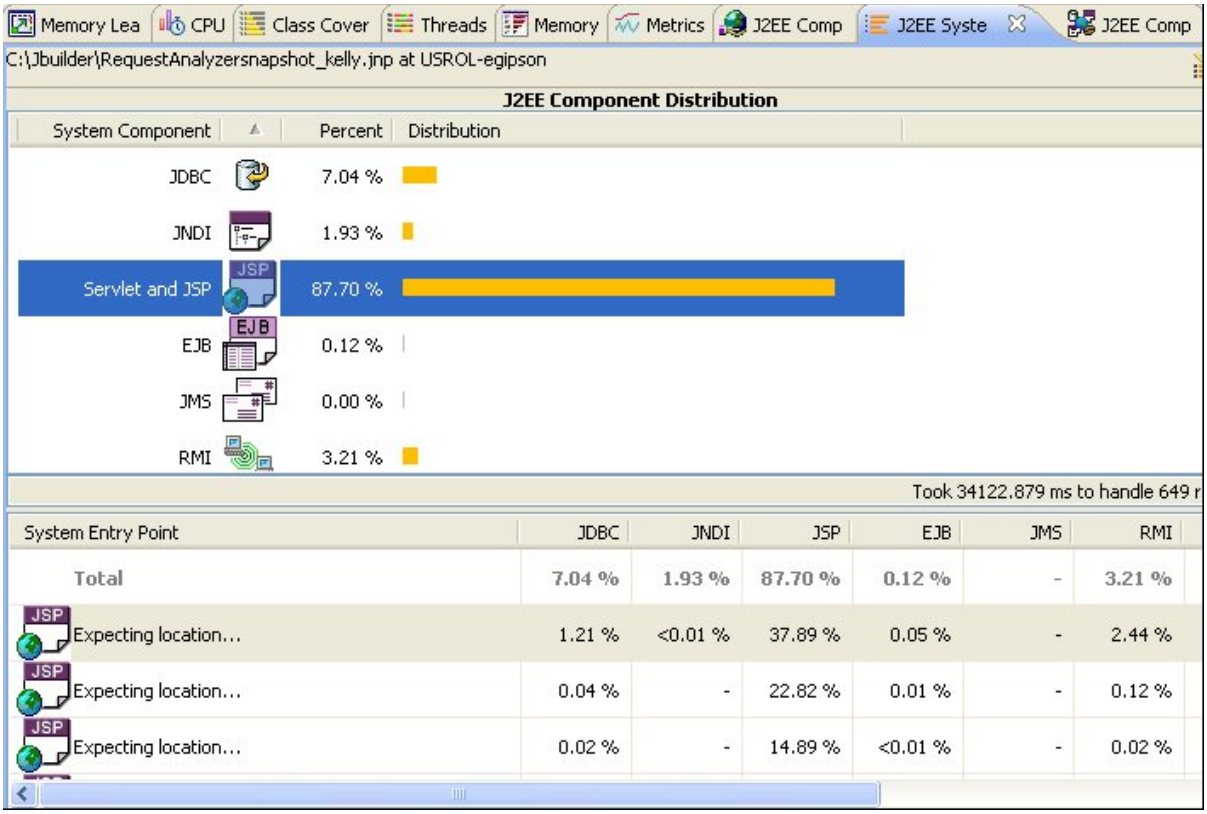
J Optimizer Request Analyzer provides two main views containing comprehensive JEE application performance data:

- The **System Dashboard** view provides a "big picture" view of JEE application performance.
- Two **JEE Component Performance** views (summary and detail)

The JEE Component Performance view provide two levels performance data: the top-level summarizes the amount of time used by each individual component, and, when you double-click on a component, a detailed view that includes graphic representation by pie-chart and standard-deviation graph. Descriptions of the System Dashboard view and the top-level JEE Component Performance view are provided below. For more information about the second-level, detailed view of individual JEE component performance, click here.

System Dashboard View

The System Dashboard provides a high level view of JEE application performance. This is the default view for J Optimizer Request Analyzer. It is a good starting point for determining where to focus your attention in analyzing performance data. A sample System Dashboard view is shown below.



The upper pane, **JEE Component Distribution,** contains a graphical representation of the amount of time associated with each of the standard JEE components (JDBC, JNDI, Servlet and JSP, EJB, JMS, RMI, CCI, and Web Services) during the profiling session. The total application activity time for the test period is listed in the lower right corner, along with the total number of requests handled by the server.

The bottom pane contains a table of **system entrypoints**. A system entrypoint is an external call into the attached application server. The table lists each entrypoint with the amount of time (as a percentage of the total application activity time) spent in each JEE component by each entry point.

Double-click an entry in the Entrypoint table to open the System Composite view. The entrypoint will be highlighted in the top pane.

JEE Component Performance Views

The JEE Component Performance views provide **two levels** of individual JEE performance data. The **top-level** displays an overview of the application time spent in each individual JEE component. When you double-click on an individual JEE component, J Optimizer opens a **second-level** JEE Component Performance view which displays more detailed information about that component, including graphic representation by pie-chart and standard-deviation graph. **For more information** about the **second-level JEE Component Performance view**, click here.

About the Top-level JEE Component Performance View

The upper pane of the top-level JEE Component Performance View is a stack trace that shows the relationship of events in terms of which events called others. The entries are ordered by amount of time used, rather than chronologically. Each event is posted with an icon and corresponding description, called a stack marker, that represents the type of action. You can look at the stack marker information in the upper section of the System Component window in several different display formats. Right-click in the upper pane to choose a display.

If you double-click any entry in the JEE Component view, J Optimizer Request Analyzer opens the appropriate component performance view for the entry. For example, double-click a tag and you will see the tag, and other JSP tags, in the Servlet and JSP Details view. To return to the System Component view, click Show System Component on the J Optimizer Request Analyzer toolbar.

The bottom pane of the JEE Component Performance View contains the following pages:

- Request Breakdown
- Hotspots
- Sub-Graph

Request Breakdown

The Request Breakdown page contains a bar graph that shows the percentage of time spent in the individual JEE components to process the selected request.Clicking a component name or the associated bar in the bar graph will open the detail view for that component. For example, clicking the bar for JDBC activity takes you to the JDBC Details view.

Hotspots

While the stack trace in the upper pane helps you understand the sequence of events, the Hotspots table in the lower pane has entries organized by how much time was taken to execute the event. The Hotspots table explains what is using time, and which are the expensive API calls. This is very useful information if you are in charge of the server, but what if you are using a URI, and four other people are using four other URIs? By default, when showing the forward tree or graph, the Hotspots table displays only those hotspots that are children of the event selected in the upper pane. This allows you to focus on your URI, with numbers based on your entries rather than on the server as a whole.

Sub-Graph

The Sub-graph page shows a graph that starts at the selected event and displays all of the calls related to that event.

Related Topics

About the JEE Quality Analyzer

Viewing Detailed JEE Performance Data

6.1. Creating a Request Analyzer Profile Configuration

The instructions on this **Help** page explain how to configure and launch the **Request Analyzer** tool using **Stand-alone J Optimizer**.

To create a new request analyzer profile configuration:

1. Open the **Profile Configuration wizard.**
2. In the left pane, choose the Java program you want to profile.
3. In the right pane, click the **Profiling** tab.
4. Use the down-arrow in the Profiling Tool field to select the **Request Analyzer** tool.
5. In the **General options** section, you can take the following actions:

Field name	Description
Virtual Machine Name	Change this name if desired. By default, J Optimizer creates a VM name based on the selected profiling tool.
Pause after launch	This option pauses the application before it executes the main method, thereby providing time to study the program's launch phase.
Pause on exit	When profiling a fast-running application, select this field to give the profiling tool time to complete data collection after the application stops running.

6. In the **Configuration overview** section, select **JEE Component details**, and any other options you want to start or enable.
7. Click **Details** to specify related settings. This generates the **Configuration Details** dialog.
8. On the **Configuration Details** dialog, in the left pane, click **Components** to expland the list of individual JEE components. This action also displays the Components settings in the right pane. **Important**: The settings in the right pane apply to *all* of the JEE components. To change the settings for an *individual* JEE component, click on the individual component's name in the left pane.
9. In the right pane, change the following settings if desired, then click **OK** to save your changes.

Field name	Description
Enable J2EE Drilldown	Keep selected to enable JEE data collection.
Application Quality Analyzer Options	The options in the Application Quality Analyzer section deliver reports on various error conditions. Deselect a report if you do not want to receive that information. For more information about the JEE Quality Analyzer reports, click here.
Error Options	Use these fields to specify a log file on Quality Analyzer results.
Miscellaneous Options/Ignore Training	This option applies only to Unix users. Select this field to ignore caching classes that you want to instrument.

10. This returns you to the **Profiling** tab, on which you can make the following additional selections:

Field name	Description

Auto-start CPU	Select to have the CPU profiler start when the Request Analyzer starts. For more information about the CPU profiler, click here.
Show VM Metrics	By default, the Request Analyzer collects VM metrics during profiling. Deselect this field if you don't want to receive these metrics. For more about VM metrics, click here.
Use Filters	J Optimizer automatically enables filter creation for use with the Request Analyzer. Deselect if you do not want to create filters. For more information about creating and using filters, click here.
Auto-Capture	Select this field to have J Optimizer generate snapshots during the Request Analyzer profiling process.
Import	Click this button to import and edit a saved profiling configuration.
Export	Click this button to save this profile configuration and reuse it later as an imported profile configuration.
Only launch the profiling agent	Select this field if you plan to view profiling results on a separate client.

11. Click **Apply** to save this profile configuration, or **Profile** to start profiling immediately.

What Happens Next

Use the J Optimizer viewing options to see and analyze the data collected by the profiling tool.

To **edit** a configuration, click on the Profile Configuration icon, then select the Profile Configurations option. This opens the wizard. In the left pane, click on the configuration, then on the tabs that contain the fields you want to change.

Related Topics

- Overview of Tool Configuration from the UI
- About the Request Analyzer Tool
- About the JEE Quality Analyzer
- Viewing JEE Performance Data

6.2. Using the JEE Quality Analyzer

You can specify use of the **Application Quality Analyzer**, a real-time validation routine used by J Optimizer to check your code for basic error conditions, when you create or edit a Request Analyzer profile configuration. The Application Quality Analyzer detects misuses of JEE, such as adding non-serializable objects to a session, forgetting to close a JDBC or JMS object, or adding large objects to a session, as well as exceptions thrown from the various JEE APIs.

Use the Application Quality Analyzer to identify the following types of problems:

- **Exceptions:** Exceptions generally indicate that an error has occurred. Any JEE exceptions thrown from JEE APIs during testing are reported in the Application Quality Analyzer view.
- **Errors:** Errors identify real problems in the application code, such as a failure to close a result set, or failure to serialize data that is supposed to be serializable.
- **Warnings:** A warning identifies code that may or may not lead to a problem. For example, the failure to close a result set before closing the statement from which the result set was generated will trigger a warning. Warnings can often provide useful information for diagnosing the cause of an error.

More information about the data collected by the JEE Application Quality Analyzer is provided in the following sections. You can select and edit these settings from within a Request Analyzer profile configuration.

Note: A different version of the Application Quality Analyzer can be launched from the CPU and Memory profile configurations.

Monitoring Session Size

The Application Quality Analyzer view is useful for QA teams, because it identifies coding errors in the application, as well as marking potential problem areas. J Optimizer Request Analyzer verifies that HttpSession entries are serializable. Failure to claim serializability in an

entry in the HttpSession will produce problems on a few application servers, and is reported as a warning. Failure to serialize an object which claims to be serializable is an error, and must be corrected. Objects which fail serialization will not failover properly in a production system, and may lead to impossible-to-reproduce production problems.

Hurried or careless development can result in applications for which the size of a serialized HttpSession may expand excessively. From pointing at an XML tree, to referencing the ServletContext within an Object in the session, it is easy to mistakenly overload an HttpSession with unnecessary information. J Optimizer Request Analyzer allows you to set a maximum threshold for session size, beyond which an error is generated. Again, as with failed serialization, a large HttpSession can lead to production instability and difficult-to-reproduce problems.

Monitoring Open Resources

JDBC and JMS activity is monitored by the Application Quality Analyzer view, as well. If your application closes a parent resource before its children are closed, a warning is reported. If the VM garbage collects a JDBC or JMS object before that object is closed, an error is reported. Some application servers automatically close JMS objects for you, and in that case, errors and warnings may not be posted. In other cases, objects which are never closed are not even available for garbage collection, and in this case, you should pay attention to the number of open resources in the # Open column of the JDBC Details view and JMS Details view. In most cases, however, the implicit close of garbage collection, which works fine on development machines, but which can fail on production systems under load, is detected. Again, errors should be considered as an indication that the application in production is likely to run into problems related to system (JDBC or JMS) resources.

How to View JEE Quality Analyzer Results

Quality Analyzer results are displayed in the **JEE Application Quality Analyzer view**. The Application Quality Analyzer view uses standard graphics to help you quickly identify the different types of reported problems. The icons in the Error column for error conditions typically include a red circle containing an exclamation point. Warning icons typically have the letter "i" in a yellow triangle. Exception icons include a red arrow. If you sort the table by the Error column, all warnings sort to the bottom of the table.

To display the JEE Application Quality Analyzer view, take the following steps:

1. Select **Windows** from the main menu.
2. Scroll down to **Show Views**, then select **JEE Application Quality Analyzer** from the context menu.

6.5. Viewing Individual JEE Component Performance Details

The J Optimizer **Request Analyzer** tool collects key performance metrics for all of the standard JEE components including JSP, JDBC, and JMS. The Request Analyzer provides two views that display the its results:

- A System Dashboard view
- Two JEE Component Performance Views (summary and detail)

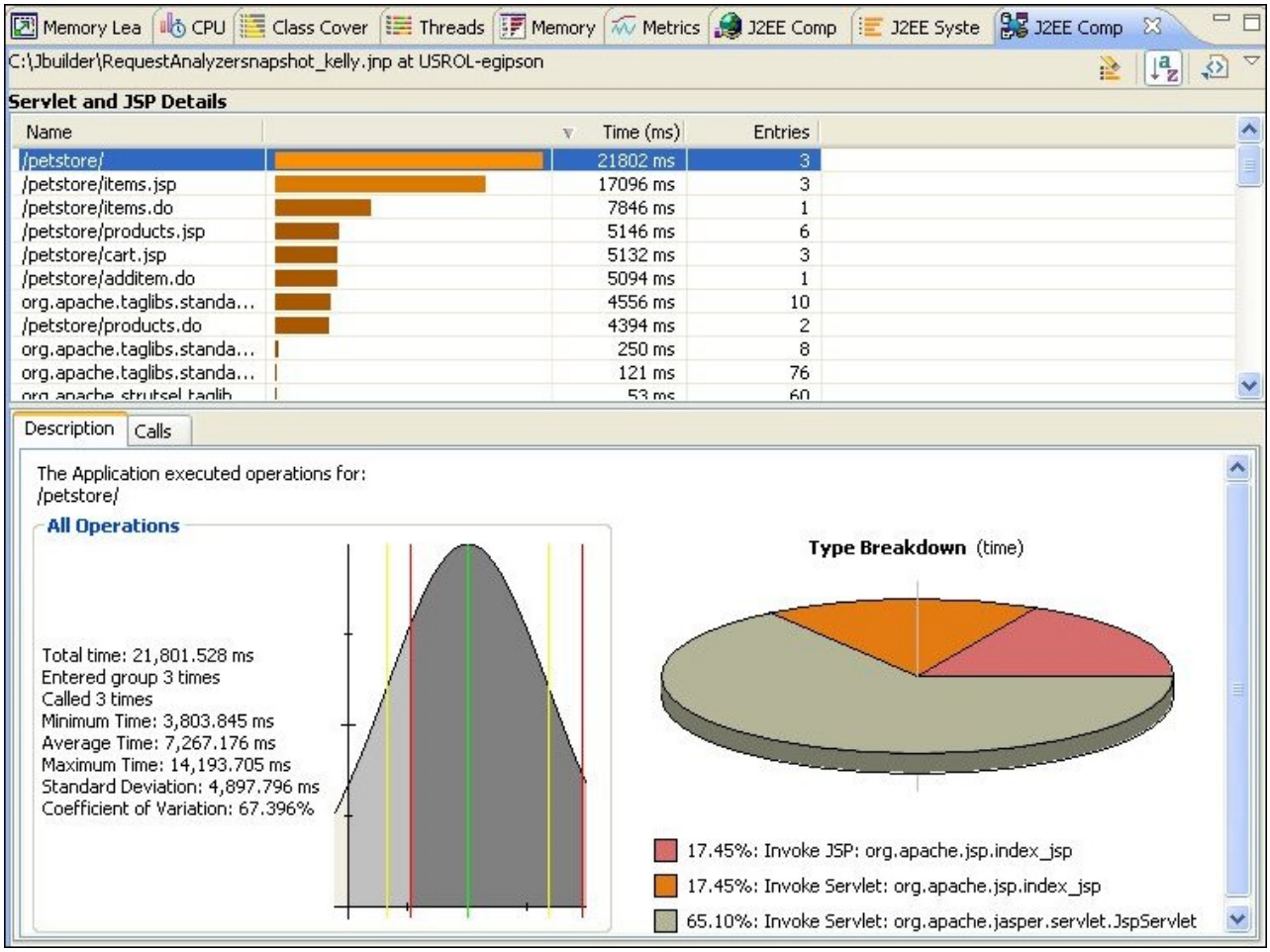
The JEE Component Performance view can display data on two levels: the top-level view summarizes the amount of time used by each event, while the second-level displays event details in graphic form. This **Help** topic explains how to view and understand data for the second-level JEE Component Performance view. For a summary of the System Dashboard view, and of the top-level JEE Component Performance view, click here.

Accessing the JEE Component Details View

The second-level JEE Component Performance view displays a graphic representation of performance activity for each individual component.

To access the second, more detailed, level:

1. In the upper pane of the **System Dashboard** view, double-click on the component for which you want to view details. For example, Servet and JSP. This opens a JEE Component Performance view tab for the component you selected.
2. In the upper pane of the JEE Component Performance view tab, double-click a line of detail. For example, as shown in the image below, /petstore/.
3. This displays component detail in two graphs in the lower pane (called "Description"). The standard deviation graph appears at left, and the pie chart at right, as shown in the image below.



About the Standard Deviation Graph

When a standard deviation is available, a graphic representation of it is displayed as an idealized bell curve. The graphic includes lines that represent the minimum and maximum values, one standard deviation from the mean, and the average (mean) value. Move the pointer over lines to see the associated values. These values are also displayed in a legend to the left of the graph.

Curves that are shifted toward the Y-axis, with a maximum line in the outlying area to the right, indicate that the majority of events were concentrated in the same area, but some events exceeded the norm. If the bell curve is too concentrated or small, you can change whether the Y-axis is forced onto the plot. Right-click on the graph, and choose Always Show Y Axis to make it easier to see the individual lines.

About the Call Breakdown Pie Chart

In addition, a breakdown of the calls by VM is displayed. By default, call data is broken down by time spent. However, you can right-click the pie chart and get a breakdown by entry count, call count, or by average time spent. If the row selected is combining types, another pie chart which separates each of the combined types is displayed. A pie chart is not available if some calls are nested, because the sum of individual nested times is greater than the whole time spent for the entry. In this case, only the legend is displayed. You can click pie pieces or legend entries to see the data for individual VMs or types. The selection, however, makes no changes in the Calls or Resources pages. Calls or Resources pages always display all data pertaining to the row selected in the upper pane.

7. Overview of Profiling from a Command Line

To profile an application or application server on a remote or offline machine, or on a machine that does not have a GUI (such as Solaris), you can configure and start **J Optimizer** from a command line.

To profile a Java program from a command line, take the following steps. Each step contains a link to more detailed instructions on how to perform that step.

1. Configure the machine, application, and/or server to work with J Optimizer. To do you, you configure the following:
    - Set environment variables.
    - Edit options as desired for each profiling tool in the `optimizeit.xml` configuration file. In addition, if you are using a socket connection, you must specify a port range in the configuration file. If you are using a TPTP connection.
    - To use a TPTP connection:

1. Run SetConfig.bat (Windows) or SetConfig.sh (Linux) to configure the J Optimizer agent-controller. When you see 'Network access mode', type ALL. This allows any host to connect to the agent.
2. Run the agent controller executable. For example, in XP run: <joptimizer-agent directory>**/bin/acserver.exe**. On Linux, run: ACStart.sh.

You are now ready to select a startup argument and profiling tool, start profiling, and use your TPTP connection to view profiling results.

2. Select the startup argument and profiling tool you want to use.
3. Attach to the J Optimizer user interface to view profiling results.

Note: If you are profiling an application that is deployed and run on an application server, you must first integrate the J Optimizer Agent with the application server. Once the J Optimizer Agent is integrated, starting the server simultaneously starts the Agent, so be sure to select the profiling tool you want to use before starting the application server.

About Remote Profiling

For remote profiling, you start both J Optimizer and the application you want to profile from the command line. You then attach to the remote program from the J Optimizer UI. This is typically used for web or enterprise applications running on a web or application server. Note that the J Optimizer Agent runs in the same VM as the application you wish to profile. It is not run separately from the application.

About Offline Profiling

J Optimizer Profiler supports offline profiling for automatically collecting and storing application performance information from a profiling session. During the profiling session, J Optimizer automatically generates snapshots at fixed intervals. Offline profiling is enabled with the

`<auto-capture>` element in the J Optimizer configuration file. Start an offline profiling session from the command line using a script or command that references the J Optimizer configuration file.

Offline profiling minimizes J Optimizer Profiler overhead in the VM because you do not attach to the application. This allows you to test an application over a long period or in a production environment. The snapshots generated during an offline profiling testing session can be compared and graphed to identify potential performance problems. View individual snapshots in J Optimizer Profiler to isolate performance problems and trace them back to their source.

For offline profiling, you start both J Optimizer Agent and the application you want to profile from the command line, with the auto-capture option enabled. This is typically used for automated testing.

Note: Offline profiling is available for J Optimizer Profiler only.

Related Topics

- Setting Environment Variables
- About Editing the J Optimizer Configuration File
- Selecting and Issuing a Startup Argument
- Setting Up the Tool Selector
- Attaching to the UI to View Profiling Results

7.1. Setting Environment Variables

Before you start a remote or offline profiling session, you must first update your environment variables to make the J Optimizer libraries available. This **Help** topic explains how to do so. Once you have set the environment variables, you can edit the configuration file as needed and start profiling.

To set environment variables for Windows:

1. Add the J Optimizer library file `optit.jar` to the CLASSPATH.
   The `optit.jar` file is located in the `lib` directory of the `J Optimizer-agent` sub-directory of your JBuilder installation. You can add `optit.jar` to the CLASSPATH from the command line, as part of a script, or with a Java command argument when you start the Agent.
   In a script or from the command line, use the following line:
   ```
   set CLASSPATH=<JBuilder2008>\joptimizer-agent\lib\optit.jar;%CLASSPATH%
   ```
   As part of the command for starting the Agent with your application server or remote application, use the following Java command option:
   ```
   -classpath <JBuilder2008>\joptimizer-agent\lib\optit.jar;%CLASSPATH%
   ```
2. Add the `J Optimizer-agent\lib` directory in your JBuilder installation to the system path.
   In a script or from the command line, use the following line to add the lib directory to the system path:

```
set PATH=<JBuilder2008>joptimizer-agent\lib;%PATH%
```

To set environment variables in Linux, Mac, and Solaris:

1. Add the J Optimizer library file `optit.jar` to the CLASSPATH.
   The `optit.jar` file is located in the `lib` directory of the `J Optimizer-agent` sub-directory of your JBuilder installation. You can add `optit.jar` to the CLASSPATH from the command line, as part of a script, or with a Java command argument when you start the Agent.
   In a script or from the command line, use the following lines:
   ```
   CLASSPATH=<JBuilder2008>/joptimizer-agent/lib/optit.jar:$CLASSPATH
   export CLASSPATH
   ```
   As part of the command for starting the Agent with your application server or remote application, use the following Java command option:
   ```
   -classpath <JBuilder2008>joptimizer-agent/lib/optit.jar:$CLASSPATH
   ```
2. Add the `lib` directory in your J Optimizer installation to the library path.
   Set this path variable:
   ```
   LD_LIBRARY_PATH=<JBuilder2008>/joptimizer-agent/lib:$LD_LIBRARY_PATH
   export LD_LIBRARY_PATH
   ```

Tip: If you are using an administrative console and your application server (such as WebSphere) does not provide ways to modify the library path, try copying the J Optimizer shared library, `liboii.so`, located in the `lib` directory of the `J Optimizer-agent` sub-directory of your JBuilder installation to the `bin` or `lib` directory of your application server. Alternatively, you may want to use symbolic links to the file in the `bin` directory of your J Optimizer installation instead of copying the file.

Related Topics

- Overview of Profiling from a Command Line
- About Editing the J Optimizer Configuration File
- Selecting and Issuing a Startup Argument
- Attaching to the Opti UI to View Profiling Results

7.2. Editing Configuration File Options

You use the **J Optimizer configuration file**, `optimizeit.xml`, to modify the profiling options available for each tool, and, when applicable, to specify a port range. The J Optimizer configuration file is used in the command to start J Optimizer and to integrate J Optimizer with application servers.

To modify configuration file options, you can create your own configuration file, or use an application server wizard to write the modified configuration file to one of your application server directories.

The J Optimizer configuration file is organized into the following sections:

- Global Options (described below)
- Memory and CPU Profiler
- Code Coverage
- Thread Debugger
- Request Analyzer

Note: Although you can set configuration options for multiple profiling tools at the same time, you can only profile with one tool at a time.

Global Options

The following element does not apply to specific J Optimizer profiling tools. This option is specified at the beginning of the file:

Element	Description
`<optimizeit-configuration>`	This is the root element for the configuration file. This element specifies the schema information for the J Optimizer configuration file.

The default J Optimizer configuration file contains the following global option settings:

```
<optimizeit-configuration xsi:noNamespaceSchemaLocation="./oiconfig.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 .
```

```
  .
  .
</optimizeit-configuration>
```

7.3. Selecting and Issuing a Startup Argument

After setting environment variables and editing the J Optimizer configuration file, you can **use one of the startup arguments** in the table below to launch both your target application and the profiling tool you want to use. When profiling completes, you can attach to the GUI to view profiling data.

The startup argument you choose must specify one of the J Optimizer Agent shared library files, which are explained in the following section. Be sure to read this section *before* deciding which library to use in the command. Your startup command should also specify a path to your bootclass files. Sample arguments are provided later in this **Help** topic.

IMPORTANT: All of the command line arguments provided on this page can be used on Windows, Mac, Linux, and Solaris machines.

Arguments	Description
`-Xrun<J Optimizer_Agent>: <path_to_J Optimizer_xml>`	**For JDK 1.5 and older VMs**. This argument specifies the name of the J Optimizer Agent shared library and the path to the J Optimizer configuration file. The J Optimizer Agent shared library and J Optimizer configuration file specify what types of data are collected during the profiling session. Required for JDK 1.5 and earlier when using JVMPI. **Note:** Do not specify the file extension in the argument for the Agent library. That is, specify -Xrunoii or -Xrunpri instead of -Xrunoii.dll or -Xrunpri.dll .
`-agentlib:<J Optimizer_Agent> =<path_to_J Optimizer_xml>`	**For JDK 1.5 and newer VMs**. This argument specifies the name of the J Optimizer Agent shared library and the path to the J Optimizer configuration file. The J Optimizer Agent shared library and J Optimizer configuration file specify what types of data are collected during the profiling session. Required for JDK 1.5 and later when using JVMTI.
`-Xbootclasspath/p: <J Optimizer_install_path>/lib/oibcp.jar`	Specifies a list of directories or JAR files to search for boot class files. Specifically, this option is used to add oibcp.jar to the BOOTCLASSPATH. The /p specifies that oibcp.jar is prepended to (added to the beginning of) the BOOTCLASSPATH.
`-classpath <J Optimizer_install_path>/lib/optit.jar`	Specifies a list of directories or JAR files to search for class files. Specifically, this option is used to add optit.jar to the CLASSPATH. **Note**: If the CLASSPATH environment variable already includes optit.jar, this option is not required.

How to Use the J Optimizer Library Files

A key component in the startup argument is the J Optimizer Agent shared library you specify. The optit.jar file, located in the J Optimizer-agent directory, contains five library files. Of these, the generic library enables use of all the J Optimizer profiling tools, while each of the other four library files launch only one specific profiling tool. The following table describes the purpose of each library.

Library Name	Library Purpose
oii	Generic library. This library enables use of all the J Optimizer profiling tools. You use this library in conjunction with the J Optimizer **Tool Selector**. Specifically, you use the Tool Selector to select the tool, then launch the command.
pri	Used to launch only the Memory and CPU Profiler.
tdi	Used to launch only the Thread Debugger tool.
cci	Used to launch only the Code Coverage tool.
j2idev	Used to launch only the Request Analyzer tool.

In most cases, we recommend specifying the generic J Optimizer Agent shared library, `oii`, instead of a tool-specific one, such as `tdi` (which launches the Thread Debugger). The generic library works in conjunction with the J Optimizer Tool Selector, which allows you to select the profiling tool you want to use before you launch the startup command.. This means that you need only to specify the generic library once in the command that runs the batch file. When you specify an individual library file in a startup argument, you must alter the batch file each time you want to run a different tool.

Note: The complete name of each library varies by operating system. For example, on Windows, the generic library file is called `oii.dll` and the Code Coverage file is called `cci.dll`. On Linux and Solaris systems, the same library files are called `liboii.so` and `libcci.so`. When you call a library file in a command, you need only use the name of the library as it is presented in the table above. Sample startup arguments are provided later in this **Help** page.

Running the Tool Selector

Before using the **generic** library in a command, you must run the J Optimizer Tool Selector. This is quickly done using the oiselector

executable. You need only run the tool selector executable once. After installing the selector, you use it to select the profiling tool you want to use, then launch the start-up command that specifies the generic library. This immediately starts your application and the profiling tool. When you want to use a different tool to profile an application on the same machine, use the Tool Selector to select it, you can again launch the same command.

For instructions on **how to run** the tool selector on Windows, Linux, Mac, and Solaris machines, click here.

IMPORTANT: You **do not** need to run or use the tool selector if your startup command specifies an indvidual tool, such as `pri` for the Memory or CPU profiler or `j2idev` for the Request Analyzer.

Sample Startup Arguments

When using JDK 1.5 or earlier, a startup argument using the **generic library** might look like the following:

```
java -Xrunoii:"C:\test_dir\j_optimizer.xml"
 -Xbootclasspath/p:"<J Optimizer_install_path>\lib\oibcp.jar"
 -classpath "<J Optimizer_install_path>\lib\optit.jar" YourJavaApp
```

The argument above starts both the Java application you specified ("YourJavaApp") and the profiling tool you selected using the J Optimizer Tool Selector.

Using a similar JDK, a startup argument using the **Request Analyzer library** file to launch the Request Analyzer might look like the following:

```
java -Xrunj2idev:"C:\test_dir\j_optimizer.xml"
 -Xbootclasspath/p:"<J Optimizer_install_path>\lib\oibcp.jar"
 -classpath "<J Optimizer_install_path>\lib\optit.jar" YourJavaApp
```

The argument above launches both the Request Analyzer and the Java program you specified ("YourJavaApp").

Note: When using the Sun Hotspot JDK, J Optimizer may require you to start your application with the `-XX:MaxPermSize` option. This option sets the permanent generation size of the virtual machine. For example, `-XX:MaxPermSize=64M` sets the permanent generation size to 64MB. If your application throws OutOfMemoryErrors when running with J Optimizer, it may be that the virtual machine is running out of permanent space, which is used to store classes. Use the `-XX:MaxPermSize` option to increase the permanent space. If you still experience problems, you can increase the Java heap size with the -Xmx option, which sets the maximum value of the Java Heap. For example, `-Xmx256M` allows the Java Heap size to grow to a maximum of 256MB.

Related Topics

- Setting Environment Variables
- About Editing the J Optimizer Configuration File
- Attaching to the UI to View Profiling Results

7.4. Setting up the Tool Selector

Before using the generic J Optimizer Agent shared library file (`oii`) in startup command, you must run the J Optimizer **Tool Selector** executable to enable tool selection. You need run this executable only once. After running the selector, you use it to select the profiling tool you want to use, then launch the startup command that specifies the generic library. This immediately starts your application and the profiling tool. When you want to use a different tool to profile an application on the same machine, use the Tool Selector to select it, then launch same command to start profiling.

Note: The tool specified with the Tool Selector is stored in a file located under the `home` directory of the current user. Run the `oiselector` command as the user who runs the application or server. Selections made with the J Optimizer Tool Selector have no affect on profiling sessions started from J Optimizer UI.

To run the tool selector and select a tool on a Windows machine:

JBuilder must be installed and the JBuilder UI opened at least once to create the joptimizer-agent directory on a Windows machine.

1. Ensure that the J Optimizer tool you want to select is not running.
2. Double-click `oiselector.exe` in the `<JBuilder2008>\joptimizer-agent\bin` directory to start the J Optimizer Tool Selector.

   An icon for the J Optimizer Tool Selector appears in the system tray. This icon indicates which tool is currently selected.

   The J Optimizer Tool Selector runs in the background.
3. Right-click the J Optimizer Tool Selector icon in the Windows system tray, and choose a tool from the context menu
4. When you have selected the tool, you can start your remote or offline profiling session.

Note: If you select **None** for the tool, then the next time you start your application server, J Optimizer will be disabled. This is particularly useful when you have configured J Optimizer with an application server started from an administrative console. This lets you enable and disable J Optimizer without changing the application server configuration.

To run the tool selector and select a tool on a Linux, Mac, or Solaris machine:

JBuilder must be installed and the JBuilder UI opened at least once to create the joptimizer-agent directory on Linux and Mac machines. A Solaris machine needs only J Optimizer-agent folder.

1. Ensure the J Optimizer tool you want to select is not running.
2. Enter `oiselector -status` in the `{J Optimizer}/bin` directory to see which tool is currently selected.
3. Enter `oiselector -config` to start an interactive configuration routine that lets you select the tool.
4. Enter the letter that corresponds to the tool you want to select.
5. When you have selected the tool, you can start your remote profiling session.

Tip: Enter `oiselector.sh -help` for a list of Tool (also referred to as" Agent") Selector command-line options and their descriptions.

Related Topics

- About Profiling from a Command Line
- Selecting and Issuing a Startup Argument
- Attaching to the J Optimizer UI to View Profiling Results

7.5. Attaching to the UI to View Profiling Data

To view the data collected by a profiling tool, you must **attach to the J Optimizer user interface**. This **Help** page provides instructions on how to do so.

IMPORTANT: If you are using J Optimizer as a JBuilder plug-in, you can attach using a socket connection or a TPTP connection. The default connection type for J Optimizer is socket. If you are using Stand-alone or Touchpoint J Optimizer, you can only use a socket connection.

Attaching with a Socket Connection

The J Optimizer profiler runs in the agent port that you specify. If you do not specify a port, the profiler defaults to 1470.

To attach to the J Optimizer agent using a socket connection:

1. Open a **Profile Configuration Wizard**.
2. Click **Attach - J/Optimizer Agent**.
3. Click the **New** button in the toolbar to create a new attach configuration.
4. On the **Host** tab, select **Attach to J/Optimizer** agent directly.
5. Select the **Profiler Type** to match the tool that you started your target application with using the oiselector tool on Solaris.
6. Enter the **IP Address/Hostname** of the remote machine.
7. Enter the **port** that was used by the J Optimizer Agent. The following table displays the range of port numbers available for each tool.

Port Number	Profiling Tool
1470 to 1480	Profiler (Memory and CPU)
1471 to 1481	Thread Debugger
1472 to 1482	Code Coverage
1473 to 1483	Request Analyzer

8. Click on **Apply** and **Profile**. Use one of the J Optimizer views associated with the profiling tool you used to view profiling results.

Attaching with a TPTP Connection

Before attaching, make sure you have configured your TPTP connection to work with J Optimizer.

To attach to the J Optimizer interface using a TPTP connection:

1. In the J Optimizer user interface, open the Profile Configuration Wizard.
2. In the left pane, select **Attach - J Optimizer Agent** and create a New configuration.
3. In the right pane, on the **Host** tab, select **Use Agent Controller to attach to J Optimizer agent**.
4. Click on the **Agents** tab.

5. Click **Refresh Data** to display a list of available agents. Select the agent you want to use and move it to the **Selected Agents** section.
6. Click **Apply** to save your selections, and **Profile** to start profiling.
7. Use one of the J Optimizer views associated with the profiling tool you used to view profiling results.

Related Topics

About Profiling from a Command Line

8. Overview of Application Server Integration

To profile an application running on an application server, you must first **integrate J Optimizer with the application server**. Integration ensures that the application server and J Optimizer run in the same virtual machine, and enables J Optimizer to capture and record information about the classes and methods loaded by applications running on the application server. Some configuration of J Optimizer is also performed as part of the ntegration process.

J Optimizer provides automated wizards which create or modify the startup scripts and configuration files required during the integration process. To profile an application server on a **Windows, Linux, or Mac** machine, you use an **agent configuration wizard** from within the J Optimizer UI. To profile an application server on a **Solaris** machine, you launch a specially designed **text-based integration wizard** from a command line. **For instructions** on how to use each wizard to profile an application server, click here.

Related Topics

How to Profile an Application Server

About Profiling From a Command Line

8.1. Profiling an Application Server

To profile an application server on a **Windows, Linux, or Mac** machine, you use an agent configuration wizard from within the J Optimizer UI. The agent configuration wizard creates a profile configuration for your application server, and integrates the server with J Optimizer. To profile an application server on a **Solaris** machine, you launch a specially designed text-based integration wizard from a command line. Once profiling completes, you can manually attach to the J Optimizer UI to view profiling results.

For an overview of how the J Optimizer wizards perform integration tasks, click here.

Note: If you want to change profiling options for any of the tools, be sure to do so before starting the procedures on this page. You can change profiling options in the `optimizeit.xml` file. Both of the wizards discussed on this page use information from that file during the integration process. For instructions on how to edit the `optimizeit.xml` file, click here.

Profiling an Application Server from the UI

To **create** an application server profile configuration from the UI:

1. Open **Window>Preferences**.
2. Navigate to **J Optimizer>Agent Configuration**.
3. In the right pane of the **Agent Configuration** window, click **Add** to add a new agent configuration.
4. On the **Add J Optimizer Agent Configuration** window, select the application server and version you want to profile. Then click **Next**.
5. Browse to and select the application server's home directory, then click **Next**.
6. The page that appears next depends on the type of application server you selected in Step 3. Skip or change this page as needed, then click **Next**.
7. On the J Optimizer Configuration page, select the **Profiling Type** (i.e. profiling tool) you want to use, then click **Edit Options** to configure the tool's settings, if desired.
8. Click **Next**. Review the **Integration Summary** page and click **Integrate**.

To **start** the Application Server:

1. In the right pane of the Agent Configuration window, click **Start**. The application server is now running with the J Optimizer agent inside it.
2. Click **OK** and look at the Console to watch application server output.
3. If you are using a socket connection, verify that the correct port number appears for the profiling tool you selected. The following table specifies the port range available for each tool.

Port Number	Profiling Tool

1470 to 1480	Profiler (Memory and CPU)
1471 to 1481	Thread Debugger
1472 to 1482	Code Coverage
1473 to 1483	Request Analyzer

You are now ready to **attach to J Optimizer to view profiling results**. For instructions on how to do so, using either a socket or a TPTP connection, click here.

Profiling an Application Server on Solaris

J Optimizer has designed a text-based integration wizard for use on Solaris machines. The text wizard displays numeric choices that represent each of the application servers J Optimizer supports. For example, 1 = Application Geronimo, 2 = Geronimo 1.1, 5 = JBoss 4.2, 11 = Sun Java AppServer 8.2, and so on for all of the supported servers. The text wizard also displays numeric choices for the J Optimizer profiling tools. For example, 1 for Profiler, or 4 for Request Analyzer.

The text-based integration wizard produces a script that you run when you want to start profiling the application server with a J Optimizer tool.

To profile an application server on Solaris:

1. Run the **integWizard.sh** file, which is located in the joptimizer-agent/bin directory.
2. When the integration text wizard options appear, select the number that corresponds to the **application server** that you want to integrate, then press Enter.
3. The next few steps vary slightly, depending on the type of application server you selected in Step 2. Generally speaking, you will be asked to provide the installation directory of the server.
4. After identifying the installation directory, select the number that represents the J Optimizer **profiling tool** you want to use. Then press Enter.
5. The wizard displays an **Integration Summary** of your selections; click **Next** to continue or **Back** to change your selections.
6. A message indicates that the integration is in progress. When it completes, click **Finish**. J Optimizer generates an integration script and locates it in the installation directory.
7. Run the newly created integrated script. J Optimizer starts profiling with the tool you selected.

You are now ready to **attach to J Optimizer to view profiling results**. For instructions on how to do so, using either a socket or a TPTP connection, click here.

Related Topics

About Application Server Integration

About Using the J Optimizer Data Views

9. Viewing and Using Data Collected by J Optimizer

The J Optimizer UI provides a variety of methods for viewing the data collected by its profiling tools. A short description of each method is provided below. For more detailed information about each methods, click on the highlighted links.

- Data Views: J Optimizer provides a number of application pages, called "views," that display data collected by the four profiling tools.
- Snapshots: Snapshots are binary files that capture all the data from a particular test run. Snapshots can be opened for analysis in the product that generated it, such as J Optimizer Profiler, J Optimizer Code Coverage, or J Optimizer Request Analyzer.
- Virtual Machine Metrics: Use the Metrics view to view the data collected by the CPU and Memory Profiler and the Request Analyzer.
- Reports: Reports that display data collected by the Memory and CPU Profiler and Request Analyzer can be generated from live data or a snapshot.
- Data Exports: You can export the data that appears in a current, open data view in an HTML or ASCII file.
- Console Output: Use this printout to browse messages from the test program or to see errors if the Java program does not start.

9.1. About Data Views

J Optimizer provides a number of ways to view the data collected by each tool. The first time you open the J Optimizer perspective, you'll see the **default views**: Memory, Memory Leak Detector, CPU, Class Coverage, Threads, and JEE Components. You can close these views, reopen them, and add additional views that were designed to work with the J Optimizer tools. The additional views include graphs that display Execution Flow, and Allocation and Thread Backtrace information. The views that are available for each tool are discussed in the following sections.

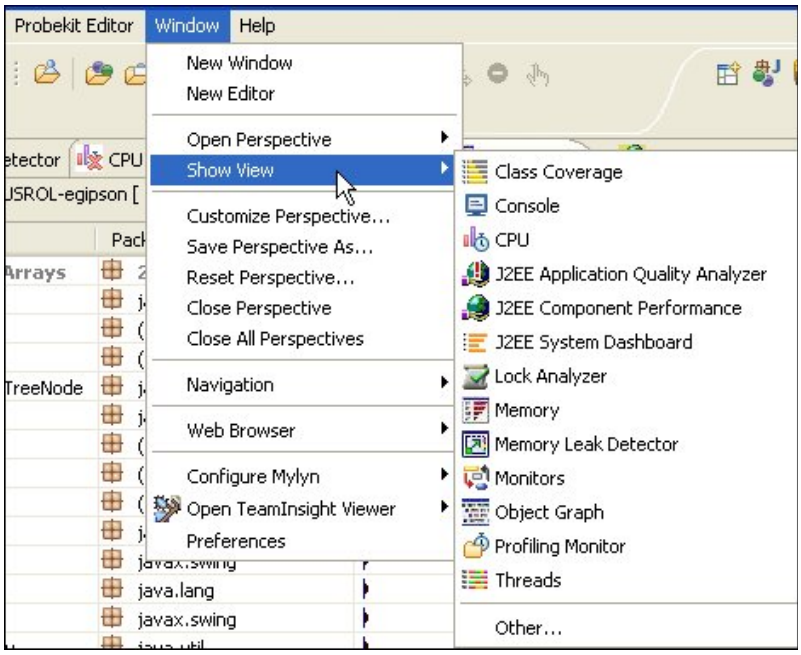
How to Access Views from the Window menu

Each view appears as a separate tab on the right side of your workspace. When a tool stops running, click on one of the view tabs to see the data the tool collected. For example, after running the memory profiler, you can click on the Memory or Memory Leak Detector tab to see memory data.

When you open J Optimizer, the views displayed are the views that were open during your last session. To **close** a view tab, click on the **X** in the right corner of the tab.

To reopen a view that you closed:

1. Open the Window menu and scroll down to **Show View**. A list of views appears, as shown below.



2. Select a view to open it as a tab in your workspace.

To see and select one of the additional views that are available:

1. Open the Window menu and scroll down to **Show Views**.
2. Select **Other**. A new Show Views window appears.
3. Scroll down the list and double-click on the **J/Optimizer folder**. A list of views appears.
4. Double-click on a view to display it as a view tab in your workspace.

Using the Profiling Monitor to Access a View

When you click on a process in the Profiling Monitor, you can see which data views are available for that process and display it in the views area. The views available for a process are determined by the tool you selected to profile it. To view and toggle to a view in the Profiling Monitor:

1. Right-click on a process and select Optimizeit from the context menu.
2. A list of available views appears. Select a view, and it will appear in the view area with the other view tabs.

Memory and CPU Profiler Views

The following views display CPU and Memory data:

- The **CPU Activities** view displays CPU utilization grouped by thread, during a specified recording interval.
- The **Memory Allocation** view displays the breakdown of the test application's Object Allocation. Click on a line in the graph to drill down to specific lines of code.
- The **Memory Leak Detector** view allows you to take heap snapshots of the application, which you can compare in order to find potential memory leaks.
- The **Object Graph** view displays the entire contents of the heap so you can view the hierarchy of objects such as busy monitors, variables, constants, and Java threads.
- The **Application Quality Analyzer** view displays a count of errors that occur during a profile session. Errors can occur when: VM pauses due to Garbage Collection exceed a specified threshold; object containers (such as StringBuffers or Byte-Arrays) exceed a

specified size; or when file descriptors are not closed.

- Use the **Metrics** view to detect signs of application problems in the following areas: Java Heap, Class Count, Thread Count, and Garbage Collection.

Code Coverage Tool Views

The **Class Coverage** view displays the lines of code per class that are covered during a profile session. Click on a line to drill down to the lines of code covered per method.

Thread Debugger Tool Views

The following views display Thread Debugger data:

- The **Threads** view displays the state of each thread during a profile session.
- The **Monitor** view identifies the locking situation of each thread during a profile session.
- The **Lock Analyzer** view identifies potential thread locking issues that may occur during a specified recording interval.

Request Analyzer Tool Views

The following views display Request Analyzer data:

- The **JEE System Dashboard** view displays a high-level view of JEE application performance.
- The **JEE Component Performance** view identifies all of the JEE events in your application, in real time, during the profile session. Provides a broad overview of the application time spent in JEE components.
- The **JEE Application Quality Analyzer** view displays a count of errors that occur during a profile session, such as exceptions thrown or when component-specific resources are not released.

9.2. Snapshots

Snapshots are binary files that capture data from a particular profiling session. You can generate snapshots in two ways: manually during a profiling session, or setting up the J Optimizer auto-capture option to capture data snapshots at timed intervals during a profiling session. You can manually generate snapshots when using the memory and CPU profiler and the Code Coverage and Request Analyzer tools. You can use the auto-capture options with the memory and CPU profiler and the Request Analyzer.

You can view open snapshots in the data view area of the J Optimizer UI.

Note: Snaphots are not backwards compatible. You cannot view snapshots produced in previous versions of J Optimizer.

To get the most from your snapshots, use the following best practices:

- Be specific: snapshots contain a lot of data. For accurate diagnostics, try to capture the snapshot at the specific point in the testing session that the application exhibits the performance problem.
- Capture baseline data: often, problems appear unexpectedly; for this reason, it is a good idea to generate and store baseline snapshots. Snapshot generation can be integrated into nightly builds to maintain a data trail. Baseline snapshots can be especially useful for comparing snapshots.
- Document your snapshots: the better you document snapshots, the easier it will be to retrieve data. When using the Generate Snapshot dialog box, specify a name that helps indicate the situation or conditions of the testing session, append the date and time stamp to the generated file, and enter comments.

9.3. About Collecting Virtual Machine Metrics

The **Virtual Machine (VM) Metrics** analyzer returns high-level, performance-related data about the program you are profiling. The Metrics view can help determine if a performance problem is related to CPU, memory, or both.

You can instruct J Optimizer to collect VM Metrics when you run the **Memory or CPU Profiler**, or the **Request Analyzer**. You specify Metrics data collection options when you create a Memory, CPU, or Request Analyzer profile configuration. Links to information about creating Memory, CPU, and Request Analyzer profile configurations are provided in the **Related Topics** section of this Help page.

Loaded Classes

The Class Count graph shows the number of classes currently loaded in the VM. This is a count of distinctly loaded classes, not necessarily

a reflection of current instance allocations. This information provides some insight into how and when classes are loaded as your program runs. For example, many applications load many JARs of classes at the time they start. Memory problems may result if the number of classes loaded continues to increase. A custom class loader may provide the ability to unload unnecessary classes to free memory.

Heap Size

Heap size is the amount of memory required to allocate Java objects. It does not include any memory allocated by the VM or by native code used by your program, classes, any thread stacks (both Java and native), or memory overhead caused by J Optimizer. The Java Heap graph plots the total heap size in the VM and the amount of heap your application is using. This information can help explain performance problems associated with garbage collection. For example, the heap size for your VM can significantly affect overall system performance. If the heap size is too large, garbage collection may occur less frequently, but full garbage collection may be very slow. If the heap size is too small, and your VM runs out of heap memory, all program execution in the VM stops until space can be freed by garbage collection.

Active Threads

The Thread Count graph shows the current number of threads running in green and the number of threads actually using some CPU resources in blue. Use this information to quickly determine whether enough threads are allocated to perform the tasks required by the test program. In general, the number of threads should match the associated application activity.

Garbage Collection

The Garbage Collection Information graph shows the garbage collector activity, which is the time spent garbage collecting divided by total time.

Related Topics

Configuring VM Metrics

9.4. Reports

The reports generated by J Optimizer tools help you share application performance data in a standard format. The data format in which reports can be generated depends on which tool you are using. Reports can be generated from live data or a snapshot.

The following J Optimizer tools can generate reports:

- Profiler
- Code Coverage

J Optimizer Profiler Reporting Features

J Optimizer Profiler generates reports as portable document format (PDF) files. The contents of the reports are based on the contents and selection in the current view. For example, if you generate a report for the Heap view, the report will reflect the column order and row selection made in the view. Reports can be generated for the following views:

- Heap view (classes loaded during testing session, and associated number of instances allocated)
- Allocation Backtraces view (data to help identify the code responsible for specific allocations)
- Quality Analyzer view (errors, warnings, and exceptions captured by the Application Quality Analyzer)

J Optimizer Code Coverage Reporting Features

J Optimizer Code Coverage generates reports as HTML or ASCII files for the current test run. The contents of J Optimizer Code Coverage reports are not based upon the content or selection in the current view. Reports can be generated from within the user interface or from the command line. At the command line, you can generate a report from a testing snapshot using the ReportGenerator class.

9.5. Exporting data

You can export the data collected by J Optimizer into an **HTML**, **XML**, or **CSV** file. After exporting the data into an HTML, XML, or CSV file, you can open it in any text viewer or browser to be printed, compared, or archived. J Optimizer provides two methods of export, both of which are explained on this **Help** page.

Note: Only views with non-graphical information support data export. You cannot export the contents of the Aggregated View - Graph display option found in the CPU Profiler view, the Memory Leak Detector view, and the Allocation Backtrace view. To capture the information in

these views, use the Hierarchical View - Tree display option.

Exporting Data from a View Tab

To export the contents of a view:

1. Click the **Export Data** icon on the view tab toolbar.

   Note: For views that do not allow export, this option is disabled.

   The **Export Report** dialog box opens.
2. Choose a format for the exported view, and then click **Next**.
   - Select **HTML** to produce an HTML document that presents data in the same format as J Optimizer displays it.
   - Select **XML** to import this data into another application.
   - Select **CSV** for a more compact file.
3. Select or enter the path of the folder where you want to save the file.
4. Enter the name of the file.
5. Click **Finish** to export the data.

After exporting the data into the specified file, J Optimizer opens the file with your default editor or web browser.

Setting Up Data Export From a Profile Configuration

The **Profile Configuration Wizard** for each profiling tool provides an **Export** option that allows you to specify a file type (CVS, HTML, XML) in which to export collected data collected. For more information, use the J Optimizer **Help** Table of Contents (TOC) to locate instructions for the profile configuration you want to use. For example, go to the "Using the Memory Profiler" section of the **Help** for instructions on creating a memory profile configuration.

9.6. Console Output

J Optimizer Profiler and J Optimizer Code Coverage both have the ability to display console messages. The Console button prints Agent messages as well as the test program standard output and standard error messages. Use this printout to browse messages from the test program or to see errors if the Java program does not start.

If you select Open A Console in the Settings dialog box when you begin your test, the standard output and the standard errors of the test program will not be redirected to the J Optimizer console.