Monitor and diagnose performance in Java SE 6

Take advantage of performance and monitoring enhancements in the latest Java release

Cathy Kegley
Greg Roberts

September 25, 2007

Java™ Platform, Standard Edition 6 (Java SE) focuses on performance, with expanded tools for managing and monitoring applications as well as diagnosing common problems. This article outlines the basis of monitoring and management in the Java SE platform and provides detailed information about the relevant enhancements in Java SE 6.

Java SE 6 provides an in-depth focus on performance, offering expanded tools for managing and monitoring applications and for diagnosing common problems. The improvements include:

- Monitoring and management API enhancements
- Official support for an improved graphical monitoring tool called JConsole
- Enhanced instrumentation of the Java virtual machine (JVM)

This article outlines the basis of monitoring and management in the Java SE platform and provides detailed information about the performance monitoring and management enhancements in the latest release. It also describes the diagnostic and troubleshooting tools available in the Java SE 6 platform.

To benefit from this article, you should have a strong understanding of the monitoring and management functionality introduced in previous Java SE releases. See Related topics for detailed background information.

Monitoring and management API

The java.lang.management package introduced in Java SE 5 defines nine MBeans called platform MBeans, or MXBeans (see Related topics). Each MXBean encapsulates a single functional area of the JVM. Beginning with Java SE 5, the JVM has included a built-in MBean server called the platform MBean server. MBeans reside in and are managed by this repository. Table 1 outlines the nine MXBeans in the Java platform:
Table 1. Platform MBeans

<table>
<thead>
<tr>
<th>Management interface</th>
<th>Resource managed</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassLoadingMXBean</td>
<td>Class loader</td>
</tr>
<tr>
<td>CompilationMXBean</td>
<td>Compiler</td>
</tr>
<tr>
<td>MemoryMXBean</td>
<td>Memory</td>
</tr>
<tr>
<td>ThreadMXBean</td>
<td>Threads</td>
</tr>
<tr>
<td>RuntimeMXBean</td>
<td>Runtime</td>
</tr>
<tr>
<td>OperatingSystemMXBean</td>
<td>Operating system</td>
</tr>
<tr>
<td>GarbageCollectorMXBean</td>
<td>Garbage collector</td>
</tr>
<tr>
<td>MemoryManagerMXBean</td>
<td>Memory manager</td>
</tr>
<tr>
<td>MemoryPoolMXBean</td>
<td>Memory pool</td>
</tr>
</tbody>
</table>

Any application can obtain and use the JVM-provided platform MBeans by obtaining an instance of the desired bean and invoking the appropriate methods. MXBeans can be used to monitor the behavior of both local and remote JVMs and retrieve information about them.

Platform MBeans provide access to information such as the number of classes loaded, uptime of the JVM, the amount of memory consumed, and the number of threads running, as well as statistics about thread contention.

You can monitor and manage the JVM resources in one of two ways:

- Direct access to the MXBean interface
- Indirect access using the MBeanServer interface

**Direct access using the MXBean interface**

You can retrieve an MXBean instance from a static factory method that gives you direct access to the locally running JVM's MXBean interface. The ManagementFactory class provides the static factory methods for obtaining the MXBeans. The example in Listing 1 demonstrates how to retrieve the RuntimeMXBean using this factory and get the value of one of its standard attributes: VmVendor:

```
Listing 1. Direct access to MXBean

RuntimeMXBean mxbean = ManagementFactory.getRuntimeMXBean();
// Get the standard attribute "VmVendor"
String vendor = mxbean.getVmVendor();
```

**Indirect access using the MBeanServer interface**

The platform MBeanServer interface uses MXBeanServerConnection to allow you to connect to remote JVMs and access MXBeans running on those platforms. You can use the ManagementFactory class's getPlatformMBeanServer method to access the platform MBean server. Listing 2 demonstrates how to get the RuntimeMXBean running in a remote JVM and get the value of its VmVendor attribute:
Listing 2. Indirect access to MXBean

```java
MBeanServerConnection serverConn;
try {
    //connect to a remote VM using JMX RMI
    JMXServiceURL url = new JMXServiceURL( "service:jmx:rmi:///jndi/rmi://<addr>");
    JMXConnector jmxConnector = JMXConnectorFactory.connect(url);
    serverConn = jmxConnector.getMBeanServerConnection();
    ObjectName objName = new ObjectName(ManagementFactory.RUNTIME_MXBEAN_NAME);

    // Get standard attribute "VmVendor"
    String vendor = (String) serverConn.getAttribute(objName, "VmVendor");
} catch (...) { }
} catch (...) { }
```

See Related topics for more detailed information on MXBeans and the java.lang.management API.

## API enhancements in Java SE 6

Java SE 5 introduced the java.util.concurrent.locks package, which provides a framework for lock and wait conditions. This framework is distinct from Java’s built-in synchronization support and allows greater flexibility in the use of locks.

Java SE 6 adds support for java.util.concurrent.locks in the java.lang.management package. This includes new classes that provide information about locks, as well as enhancements to the ThreadInfo, ThreadMXBean, and OperatingSystemMXBean interfaces.

Java SE 6 introduces two new classes:

- **LockInfo** contains information about a lock.
- **MonitorInfo** extends LockInfo and contains information about an object-monitor lock.

The ThreadInfo class makes use of these new objects with the introduction of three new methods:

- `getLockInfo()` returns the LockInfo object for which the given thread is blocked waiting.
- `getLockedMonitors()` returns the MonitorInfo objects that are currently locked by the given thread.
- `getLockedSynchronizers()` returns the LockInfo objects, representing ownable synchronizers that are currently locked by the given thread.

In Java SE 5, the ThreadMXBean.getThreadInfo methods report only an object monitor that a thread is waiting to acquire or is blocked from entering. In Java SE 6, these methods are enhanced to report the AbstractOwnableSynchronizer that a thread is waiting to acquire.

Four new methods have been added to the ThreadMXBean interface:

- `isObjectMonitorUsageSupported()` tests if the virtual machine supports monitoring the usage of object monitors.
• `isSynchronizerUsageSupported()` tests if the virtual machine supports monitoring the usage of ownable synchronizers.

• `findDeadlockedThreads()` returns an array of thread IDs that are deadlocked. Threads that are deadlocked are blocking one another from entering an object monitor or synchronizer.

• `dumpAllThreads()` returns stack-trace and synchronization information for all live threads.

Finally, the `OperatingSystemMXBean` interface was updated to include the `getSystemLoadAverage()` method, which returns the system load average for the past minute.

In addition to this programmatic support, Java SE 6 also includes several diagnostic and troubleshooting tools that can be used to detect problems and monitor usage of JVM resources. The next two sections describe and demonstrate some of the available diagnostic tools.

### Java Monitoring and Management Console (JConsole)

Java SE 6 includes official support for JConsole, a monitoring and management console introduced in Java SE 5. JConsole lets you monitor various JVM resource statistics during runtime. It’s particularly useful for detecting symptoms of deadlocks, lock contention, memory leaks, and cycling threads. It can connect to a local or remote JVM and can be used to monitor:

- Thread state (including associated locks)
- Memory usage
- Garbage collection
- Runtime information
- JVM information

The following subsections describe the enhancements made to JConsole in Java SE 6. See [Related topics](#) for more information on how to start and use JConsole.

### Attach API support

Beginning in Java SE 6, JConsole implements the new Attach API. This API consists of two packages — `com.sun.tools.attach` and `com.sun.tools.attach.spi` — that let implementing applications dynamically attach to a target virtual machine and run their agents within that JVM.

In the past, applications that you wanted to monitor with JConsole needed to be started with the `-Dcom.sun.management.jmxremote` option; applications no longer need to start with this option. Support for dynamic attachment makes JConsole capable of monitoring any application that supports the Attach API. Compliant applications are automatically detected when JConsole starts up.

### Enhanced UI and MBean presentation

In Java SE 6, JConsole has been updated to have a look and feel similar to the Windows® operating system or GNOME desktop, depending on which platform it’s running on. The screenshots shown throughout the rest of this article were taken on Windows XP and show the UI features that have changed from the previous release.
Once started and associated with an application, the JConsole view consists of six tabs, each representing a different JVM resource or set of resources:

- Overview
- Memory
- Threads
- Classes
- VM Summary
- MBeans

The Overview tab displays correlated information about memory usage, threads, classes, and CPU usage in a graphical format. The Overview tab displays a set of related information on one page that was available previously only by switching among multiple tabs. Figure 1 shows the Overview tab for a sample application:
Figure 1. JConsole Overview tab

The Overview tab displays four graphs of VM resource-usage information as well as a pick list for altering the time range for which you would like to see results. The first graph, Heap Memory...
Usage, displays the amount of heap memory that has been used in megabytes over time. This graph is useful in detecting memory leaks. If a memory leak is present in your application, the heap memory usage steadily increases over time.

The Threads graph plots the number of live threads over time, and the Classes graph depicts the number of classes loaded. The CPU Usage chart depicts the percentage of the CPU your application uses at various points in its life cycle.

The VM Summary tab, shown in Figure 2, is another new addition to the Java SE 6 release. It provides detailed information about the JVM, including total uptime, threading information, classes loaded, memory statistics, garbage collection, and operating-system information.
The MBeans tab has been improved to allow for easier access to your MBeans' operations and attributes. It displays information about all MBeans registered with the platform. All platform
MBeans are accessible through this tab. The tree structure along the left-hand side displays all currently running MBeans. When you select an MBean, its MBeanInfo and descriptor are displayed on the table to the right, as shown in Figure 3:

**Figure 3. JConsole MBean tab**

Selecting the Attributes node displays all the MBean's attributes, as shown in Figure 4 for the Threading MBean:
Figure 4. MBean attributes

Note that the attributes and their values shown in the box to the right map to the attribute values attainable through the ThreadMXBean API in the java.lang.management package previously described. You can obtain additional information about a listed attribute by double-clicking on the attribute value. Only attribute values shown in bold can be expanded. For example, double-clicking on the AllThreadIds value displays the thread IDs of all 22 threads, as shown in Figure 5:
Writeable attributes are displayed in blue and you can edit them by clicking on them and entering the new value. For example, the `ThreadContentionMonitoringAvailable` attribute shown in Figure 5 can be edited in this manner from the JConsole view.

Selecting the Operations node in the left-hand tree structure displays the operations associated with that MBean. The MBean operations appear as buttons in the right-side display and, when clicked on, invoke the specified method. Figure 6 shows the operations available for `ThreadMXBean`:
The HotSpot Diagnostic MBean

In Java SE 6, JConsole includes support for the HotSpot Diagnostic MBean. This MBean was introduced in this release to allow you to perform on-the-spot diagnostic operations. Its API lets users perform a heap dump and set other VM options during run time. You can access the HotSpot Diagnostic MBean from the MBean tab by expanding the `com.sun.management` node and selecting `HotSpotDiagnostic`. Methods available from the HotSpot Diagnostic MBean are shown in Figure 7:
JConsole plug-in support

Beginning in Java SE 6, JConsole includes plug-in support that allows you to build your own plug-ins to run with JConsole. For example, you can add a custom tab to the JConsole main view for accessing application-specific MBeans and for performing your own monitoring activities.

You must extend the abstract `com.sun.tools.jconsole.JConsolePlugin` class to create a custom JConsole plug-in. You implement two methods for a plug-in to appear properly in the JConsole view:

- `newSwingWorker()` returns a `SwingWorker` object that performs the GUI updates for your plug-in.
- `getTabs()` returns a map of tabs to be added to the JConsole window.

JConsole uses its service-provider mechanism to detect and load all plug-in classes. For this reason, you must provide your plug-in class in a JAR file containing a file named `META-INF/services/com.sun.tools.jconsole.JConsolePlugin`. This file should contain a list of the fully qualified plug-in class names, one per line. To load the new plug-ins into the JConsole view, run JConsole from the command line with the command:

```
jconsole -pluginpath plugin_path
```

In this command, `plugin_path` refers to the paths to the directory or archive of JConsole plug-ins. You can specify multiple paths.
Java SE 6 comes equipped with a sample JConsole plug-in called JTop. JTop shows the CPU usage of the threads running within the current application. To run JConsole with the JTop, execute the command:

```
jconsole -pluginpath JAVA_HOME/demo/management/JTop/JTop.jar
```

Figure 8 shows an instance on JConsole with the JTop tab selected. The left-hand column displays the names of all running threads. For each thread, the CPU usage and thread state are displayed. This view refreshes automatically as statistics change. The JTop plug-in is useful for identifying threads with high CPU consumption.

**Figure 8. JConsole JTop plug-in**

![JTop view](image)

# Monitoring and troubleshooting tools

In addition to JConsole, Java SE 6 includes support for a number of other command-line tools. These diagnostic tools can attach to any application without requiring that application to start in a special mode. They enable you to obtain more information about an application to determine if it is behaving as you expect. Note that these tools are listed as experimental and might not be fully supported in future Java SE releases.

## Monitoring tools

Java SE 6 includes three command-line utilities, listed in Table 2, that are useful for monitoring JVM performance statistics:

**Table 2. Monitoring tools**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jps</td>
<td>JVM process status tool</td>
</tr>
<tr>
<td>jstat</td>
<td>JVM statistics monitoring tool</td>
</tr>
<tr>
<td>jstatd</td>
<td>JVM jstat daemon</td>
</tr>
</tbody>
</table>
The `jps` utility lists the virtual machines for the current user on the target system. The utility is useful in environments where the VM is started using the JNI Invocation API rather than the standard Java launcher. In these environments, it is not always easy to recognize the Java processes in the process list. The `jps` tool alleviates this problem.

The following example demonstrates the use of the `jps` utility. Simply enter `jps` on the command line, and the utility lists the virtual machines and process IDs for which the user has access rights, as shown in the example in Listing 3:

**Listing 3. Using the jps utility**

```bash
$ jps
16217 MyApplication
16342 jps
```

The `jstat` utility uses the JVM's built-in instrumentation to provide information on performance and resource consumption of running applications. The tool is useful for diagnosing performance issues, particularly issues related to heap sizing and garbage collection.

The `jstatd` daemon is a Remote Method Invocation (RMI) server application that monitors the creation and termination of JVMs and provides an interface to allow remote monitoring tools to attach to JVMs running on the local host. For example, this daemon allows the `jps` utility to list processes on a remote system.

See [Related topics](#) for additional documentation and usage examples for each of these tools.

**Troubleshooting tools**

Java SE 6 also includes a number of troubleshooting tools, listed in Table 3, that can help you pinpoint portions of your application that are behaving unexpectedly:

**Table 3. Troubleshooting tools**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jinfo</td>
<td>Configuration information</td>
</tr>
<tr>
<td>jhat</td>
<td>Heap dump browser</td>
</tr>
<tr>
<td>jmap</td>
<td>Memory map</td>
</tr>
<tr>
<td>jsadebugd</td>
<td>Serviceability agent debug daemon</td>
</tr>
<tr>
<td>jstack</td>
<td>Stack trace</td>
</tr>
</tbody>
</table>

The `jinfo` command-line utility retrieves configuration information from a running Java process or crash dump and prints the system properties or the command-line flags that were used to start the virtual machine.

The `jhat` tool provides a convenient means to browse the object topology in a heap snapshot. This tool, introduced in Java SE 6 release to replace the Heap Analysis Tool (HAT), is useful in detecting memory leaks.
The `jmap` command-line utility prints memory-related statistics for a running VM or core file. The utility can also use the `jsadebugd` daemon to query a process or core file on a remote machine. The `jmap` tool is useful in diagnosing excessive use of finalizers, which can result in an `OutOfMemoryError`.

The Serviceability Agent Debug Daemon (`jsadebugd`) attaches to a Java process or to a core file and acts as a debug server. This utility is currently available only on Solaris OS and Linux®. Remote clients such as `jstack`, `jmap`, and `jinfo` can attach to this server using Java RMI.

The `jstack` command-line utility attaches to the specified process or core file and prints the stack traces of all threads that are attached to the virtual machine, including Java threads and VM internal threads, and optionally native stack frames. The utility also performs deadlock detection. It can use the `jsadebugd` daemon to query a process or core file on a remote machine. The `jstack` tool is useful for diagnosing deadlocks.

See Related topics for additional documentation and usage examples for each of these tools.

**Conclusion**

The Java 6 platform delivers several enhancements in VM instrumentation, management APIs, and JDK tools to help you efficiently identify and diagnose performance and memory problems within Java applications. This article describes the improvements made to the Java SE monitoring and managing framework and touches on the diagnostic command-line utilities available to developers.

Average Java application speed has steadily increased over time. Now, with the Java SE 6 release, Java performance is comparable to that of C or C++. In many cases, Java code runs significantly faster. And you can use the tools described here to achieve better performance optimization. Give them a try. We guarantee you'll find places to optimize your application where you never knew you could.
Related topics

- "Monitoring and Managing Java SE 6 Platform Applications" (Mandy Chung, Sun Developer Network, August 2006): How to diagnose common performance problems in Java SE.
- **java.lang.management API**: Take a look at this API for your Java monitoring and management development needs.
- **JDK Tools and Utilities**: Check out Sun's site for the utilities included in Java SE 6.
- **jps, jstat, jstatd, jinfo, jhat, jmap, jsaddebug, and jstack**: Documentation and examples for the monitoring and troubleshooting tools introduced in this article.

© Copyright IBM Corporation 2007

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)