



# Java SE Monitoring and Management Guide



Sun Microsystems, Inc.  
4150 Network Circle  
Santa Clara, CA 95054  
U.S.A.

Part No: N/A  
October 2006

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents or pending patent applications in the U.S. and in other countries.

U.S. Government Rights – Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, the Solaris logo, the Java Coffee Cup logo, docs.sun.com, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

Products covered by and information contained in this publication are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical or biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Sun Microsystems, Inc. détient les droits de propriété intellectuelle relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plusieurs brevets américains ou des applications de brevet en attente aux Etats-Unis et dans d'autres pays.

Cette distribution peut comprendre des composants développés par des tierces personnes.

Certaines composants de ce produit peuvent être dérivées du logiciel Berkeley BSD, licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays; elle est licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, le logo Solaris, le logo Java Coffee Cup, docs.sun.com, Java et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui, en outre, se conforment aux licences écrites de Sun.

Les produits qui font l'objet de cette publication et les informations qu'il contient sont régis par la législation américaine en matière de contrôle des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes chimiques ou biologiques ou pour le nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers des pays sous embargo des Etats-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régis par la législation américaine en matière de contrôle des exportations et la liste de ressortissants spécifiquement désignés, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.

# Contents

---

<b>Preface</b> .....	7
<b>1 Overview of Java SE Monitoring and Management</b> .....	11
Key Monitoring and Management Features .....	11
Java VM Instrumentation .....	11
Monitoring and Management API .....	12
Monitoring and Management Tools .....	12
Java Management Extensions (JMX) Technology .....	13
Platform MXBeans .....	14
Platform MBean Server .....	15
<b>2 Monitoring and Management Using JMX Technology</b> .....	17
Setting System Properties .....	17
Enabling the Out-of-the-Box Management .....	18
Local Monitoring and Management .....	18
Remote Monitoring and Management .....	19
Using Password and Access Files .....	24
Password Files .....	24
Access Files .....	25
Out-of-the-Box Monitoring and Management Properties .....	26
Configuration Errors .....	28
Connecting to the JMX Agent Programmatically .....	29
Setting up Monitoring and Management Programmatically .....	29
Mimicking Out-of-the-Box Management Using the JMX Remote API .....	30
Example of Mimicking Out-of-the-Box Management .....	31
<b>3 Using JConsole</b> .....	37
Starting JConsole .....	37

---

Command Syntax .....	37
Connecting to a JMX Agent .....	39
Presenting the JConsole Tabs .....	43
Viewing Overview Information .....	44
Monitoring Memory Consumption .....	45
Monitoring Thread Use .....	48
Monitoring Class Loading .....	50
Viewing VM Information .....	51
Monitoring and Managing MBeans .....	53
Creating Custom Tabs .....	63
<b>4 Using the Platform MBean Server and Platform MXBeans .....</b>	<b>65</b>
Using the Platform MBean Server .....	65
Accessing Platform MXBeans .....	65
Accessing Platform MXBeans via the ManagementFactory Class .....	66
Accessing Platform MXBeans via an MXBean Proxy .....	66
Accessing Platform MXBeans via the MBeanServerConnection Class .....	67
Using Sun Microsystems' Platform Extension .....	67
Accessing MXBean Attributes Directly .....	67
Accessing MXBean Attributes via MBeanServerConnection .....	68
Monitoring Thread Contention and CPU Time .....	69
Managing the Operating System .....	69
Logging Management .....	70
Detecting Low Memory .....	70
Memory Thresholds .....	71
Polling .....	72
Threshold Notifications .....	73
<b>5 SNMP Monitoring and Management .....</b>	<b>75</b>
Enabling the SNMP Agent .....	75
Access Control List File .....	75
▼ To Enable the SNMP Agent in a Single-user Environment .....	76
▼ To Enable the SNMP Agent in a Multiple-user Environment .....	76
SNMP Monitoring and Management Properties .....	77
Configuration Errors .....	78

- A Additional Security Information For Microsoft Windows** ..... 79
  - How to Secure a Password File on Microsoft Windows Systems ..... 79
    - ▼ To Secure a Password File on Windows XP Professional Edition ..... 79
    - ▼ To Secure a Password File on Windows XP Home Edition ..... 86



# Preface

---

The Java Platform, Standard Edition (Java SE platform) 6 features utilities that allow you to monitor and manage the performance of a Java Virtual Machine (Java VM) and the Java applications that are running in it. The *Java SE Monitoring and Management Guide* describes those monitoring and management utilities.

## Who Should Use This Book

The *Java SE Monitoring and Management Guide* is intended for experienced users of the Java language, such as systems administrators and software developers, for whom the performance of the Java platform and their applications is of vital importance.

## Before You Read This Book

It is recommended that users are familiar with several other features of the Java SE platform. The following documentation might be of use.

- [Java Management Extensions \(JMX\) Technology](#)
- [Java HotSpot Technology](#)
- [Java Virtual Machine Technology](#)
- [Performance Documentation for the Java HotSpot VM](#)

## How This Book Is Organized

This book covers the following topics.

- [Chapter 1](#) introduces the monitoring and management utilities provided with the Java SE platform.
- [Chapter 2](#) describes how to configure your platform to allow monitoring and management using the JMX API.
- [Chapter 3](#) introduces the JConsole graphical user interface.
- [Chapter 4](#) presents the MBeans that are provided with the Java SE platform for monitoring and management purposes.

- [Chapter 5](#) explains how to perform monitoring and management using the Simple Network Management Protocol (SNMP).
- [Appendix A](#) provides security configuration information specific to Windows platforms.

## Related Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

---

**Note** – Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

---

## Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- [Documentation](http://www.sun.com/documentation/) (<http://www.sun.com/documentation/>)
- [Support](http://www.sun.com/support/) (<http://www.sun.com/support/>)
- [Training](http://www.sun.com/training/) (<http://www.sun.com/training/>)

## Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P-1 Typographic Conventions

Typeface	Meaning	Example
AaBbCc123	The names of commands, files, and directories, and onscreen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>machine_name% you have mail.</code>
<b>AaBbCc123</b>	What you type, contrasted with onscreen computer output	<code>machine_name% su</code> Password:
<i>aabbcc123</i>	Placeholder: replace with a real name or value	The command to remove a file is <code>rm filename.</code>



TABLE P-1 Typographic Conventions (Continued)

Typeface	Meaning	Example
<i>AaBbCc123</i>	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the <i>User's Guide</i> . A <i>cache</i> is a copy that is stored locally. Do <i>not</i> save the file. <b>Note:</b> Some emphasized items appear bold online.

## Shell Prompts in Command Examples

The following table shows the default UNIX® system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P-2 Shell Prompts

Shell	Prompt
C shell	machine_name%
C shell for superuser	machine_name#
Bourne shell and Korn shell	\$
Bourne shell and Korn shell for superuser	#



# Overview of Java SE Monitoring and Management

---

This chapter introduces the features and utilities that provide monitoring and management services to the Java Platform, Standard Edition (Java SE platform). The features introduced here will be expanded upon in the following chapters of this guide.

## Key Monitoring and Management Features

The Java SE platform includes significant monitoring and management features. These features fall into four broad categories.

- Instrumentation for the Java Virtual Machine (Java VM).
- Monitoring and Management application programming interfaces (API).
- Monitoring and Management tools.
- The Java Management Extensions (JMX) technology.

These categories of monitoring and management features are introduced further in the next sections.

## Java VM Instrumentation

The Java VM is instrumented for monitoring and management, enabling built-in (or *out-of-the-box*) management capabilities that can be accessed both remotely and locally. For more information, see [Chapter 2](#) and [Chapter 5](#).

The Java VM includes a platform MBean server and platform MBeans for use by management applications that conform to the JMX specification. These are implementations of the monitoring and management API described in the next section. Platform MXBeans and MBean servers are introduced in “[Platform MXBeans](#)” on page 14 and “[Platform MBean Server](#)” on page 15.

Example code is provided in the `JDK_HOME/demo/management` directory, where `JDK_HOME` is the directory in which the Java Development Kit (JDK) is installed.

## Monitoring and Management API

The `java.lang.management` package provides the interface for monitoring and managing the Java VM. This API provides access to the following types of information.

- Number of classes loaded and threads running.
- Java VM uptime, system properties, and VM input arguments.
- Thread state, thread contention statistics, and stack trace of live threads.
- Memory consumption.
- Garbage collection statistics.
- Low memory detection.
- On-demand deadlock detection.
- Operating system information.

In addition to the `java.lang.management` API, the `java.util.logging.LoggingMXBean` API provides allows you to perform monitoring and management of logging.

## Monitoring and Management Tools

The Java SE platform provides a graphical monitoring tool called JConsole. JConsole implements the JMX API and enables you to monitor the performance of a Java VM and any instrumented applications, by providing information to help you optimize performance. Introduced in the J2SE platform 5.0, JConsole became an officially supported feature of the platform in the Java SE platform, version 6.

Some of the enhancements that have been made to JConsole between these two releases of the Java SE platform are as follows.

- JConsole Plug-in support, that allows you to build your own plug-ins to run with JConsole, for example, to add a custom tab for accessing your applications' MBeans.
- Dynamic attach capability, allowing you to connect JConsole to any application that supports the Attach API, that was added to the Java SE platform, version 6.
- Enhanced user interface, that makes data more easily accessible.
- New Overview and VM Summary tabs, for a better presentation of general information about your Java VM.
- The HotSpot Diagnostic MBean, which provides an API to request heap dump at runtime and also change the setting of certain VM options.
- Improved presentation of MBeans, to make it easier to access your MBeans' operations and attributes.

JConsole is presented in full in [Chapter 3](#).

Other command-line tools are also supplied with the Java SE platform. See the [Monitoring Tools](#) section of the JDK Development Tools document for more information.

## Java Management Extensions (JMX) Technology

The Java SE platform, version 6 includes the JMX specification, version 1.4. The JMX API allows you to instrument applications for monitoring and management. An RMI connector allows this instrumentation to be remotely accessible, for example by JConsole.

For more information, see the [JMX technology documentation](#) for the Java SE platform. A very brief introduction to the main components of the JMX API is included in the next sections.

### What are MBeans?

JMX technology MBeans are *managed beans*, namely Java objects that represent resources to be managed. An MBean has a *management interface* consisting of the following.

- Named and typed attributes that can be read and written.
- Named and typed operations that can be invoked.
- Typed notifications that can be emitted by the MBean.

For example, an MBean representing an application's configuration could have attributes representing different configuration parameters, such as a cache size. Reading the `CacheSize` attribute would return the current size of the cache. Writing `CacheSize` would update the size of the cache, potentially changing the behavior of the running application. An operation such as `save` could store the current configuration persistently. The MBean could send a notification such as `ConfigurationChangedNotification` when the configuration changes.

MBeans can be standard or dynamic. Standard MBeans are Java objects that conform to design patterns derived from the JavaBeans component model. Dynamic MBeans define their management interface at runtime. More recently, an additional type of MBean called an `MXBean` has also been added to the Java platform.

- A *standard MBean* exposes the resource to be managed directly through its attributes and operations. Attributes are exposed through "getter" and "setter" methods. Operations are the other methods of the class that are available to managers. All these methods are defined statically in the MBean interface and are visible to a JMX agent through introspection. This is the most straightforward way of making a new resource manageable.
- A *dynamic MBean* is an MBean that defines its management interface at runtime. For example, a configuration MBean could determine the names and types of the attributes it exposes by parsing an XML file.
- An *MXBean* is a new type of MBean that provides a simple way to code an MBean that only references a predefined set of types. In this way, you can be sure that your MBean will be usable by any client, including remote clients, without any requirement that the client have access to model-specific classes representing the types of your MBeans. The platform MBeans introduced below are all MXBeans.

## MBean Server

To be useful, an MBean must be registered in an MBean server. An MBean Server is a repository of MBeans. Each MBean is registered with a unique name within the MBean server. Usually the only access to the MBeans is through the MBean server. In other words, code does not access an MBean directly, but rather accesses the MBean by name through the MBean server.

The Java SE platform includes a built-in platform MBean server. For more information, see [Chapter 4](#).

## Creating and Registering MBeans

There are two ways to create an MBean. One is to construct a Java object that will be the MBean, then use the `registerMBean` method to register it in the MBean Server. The other is to create and register the MBean in a single operation using one of the `createMBean` methods.

The `registerMBean` method is simpler for local use, but cannot be used remotely. The `createMBean` method can be used remotely, but sometimes requires attention to class loading issues. An MBean can perform actions when it is registered in or unregistered from an MBean Server if it implements the `MBeanRegistration` interface.

## Instrumenting Applications

General instructions on how to instrument your applications for management by the JMX API is beyond the scope of this document. See the documentation for the [Java Management Extensions \(JMX\) Technology](#) for information.

# Platform MBeans

A platform MXBean is an MBean for monitoring and managing the Java VM and other components of the Java Runtime Environment (JRE). Each MXBean encapsulates a part of VM functionality such as the class loading system, just-in-time (JIT) compilation system, garbage collector, and so on.

[Table 1-1](#) lists all the platform MXBeans and the aspect of the VM that they manage. Each platform MXBean has a unique `javax.management.ObjectName` for registration in the platform MBean server. A Java VM may have zero, one, or more than one instance of each MXBean, depending on its function, as shown in the table.

TABLE 1-1 Platform MBeans

Interface	Part of VM Managed	Object Name	Instances per VM
<code>ClassLoaderMXBean</code>	Class loading system	<code>java.lang:type=ClassLoader</code>	One

TABLE 1-1 Platform MXBeans (Continued)

Interface	Part of VM Managed	Object Name	Instances per VM
CompilationMXBean	Compilation system	java.lang:type=Compilation	Zero or one
GarbageCollectorMXBean	Garbage collector	java.lang:type=GarbageCollector, name=collectorName	One or more
LoggingMXBean	Logging system	java.util.logging:type=Logging	One
MemoryManagerMXBean (sub-interface of GarbageCollectorMXBean)	Memory pool	java.lang: typeMemoryManager, name=managerName	One or more
MemoryPoolMXBean	Memory	java.lang: type= MemoryPool, name=poolName	One or more
MemoryMXBean	Memory system	java.lang:type= Memory	One
OperatingSystemMXBean	Underlying operating system	java.lang:type= OperatingSystem	One
RuntimeMXBean	Runtime system	java.lang:type= Runtime	One
ThreadMXBean	Thread system	java.lang:type= Threading	One

See the package description in the API reference for the [java.lang.management](#) package for details of the platform MXBeans (apart from LoggingMXBean). See the API reference for [java.util.logging](#) for details of the LoggingMXBean.

## Platform MBean Server

The *platform MBean Server* can be shared by different managed components running within the same Java VM. You can access the platform MBean Server with the method `ManagementFactory.getPlatformMBeanServer()`. The first call to this method, creates the platform MBean server and registers the platform MXBeans using their unique object names. Subsequently, it returns the initially created platform MBeanServer instance.

MXBeans that are created and destroyed dynamically (for example, memory pools and managers) will automatically be registered and unregistered in the platform MBean server. If the system property `javax.management.builder.initial` is set, the platform MBean server will be created by the specified `MBeanServerBuilder`.

You can use the platform MBean server to register other MBeans besides the platform MXBeans. This enables all MBeans to be published through the same MBean server and makes network publishing and discovery easier.



# Monitoring and Management Using JMX Technology

---

The Java virtual machine (Java VM ) has built-in instrumentation that enables you to monitor and manage it using the Java Management Extensions (JMX) technology. These built-in management utilities are often referred to as *out-of-the-box management* tools for the Java VM. You can also monitor any appropriately instrumented applications using the JMX API.

## Setting System Properties

To enable and configure the out-of-the-box JMX agent so that it can monitor and manage the Java VM, you must set certain system properties when you start the Java VM. You set a system property on the command-line as follows.

```
java -Dproperty=value ...
```

You can set any number of system properties in this way. If you do not specify a value for a management property, then the property is set with its default value. The full set of out-of-the-box management properties is described in [Table 2-1](#) at the end of this chapter. You can also set system properties in a configuration file, as described in “[Out-of-the-Box Monitoring and Management Properties](#)” on page 26.

---

**Note** – To run the Java VM from the command line, you must add `JRE_HOME/bin` to your path, where `JRE_HOME` is the directory containing the Java Runtime Environment (JRE) implementation. Alternatively, you can enter the full path when you type the command.

---

The following documents describe the syntax and the full set of command-line options supported by the Java HotSpot VMs.

- [Java application launcher for Microsoft Windows](#)
- [Java application launcher for Solaris Operating Environment](#)
- [Java application launcher for Linux](#)

# Enabling the Out-of-the-Box Management

To monitor a Java platform using the JMX API, you must do the following.

1. Enable the JMX agent (another name for the platform MBean server) when you start the Java VM. You can enable the JMX agent for:
  - Local monitoring, for a client management application running on the local system.
  - Remote monitoring, for a client management application running on a remote system.
2. Monitor the Java VM with a tool that complies to the JMX specification, such as JConsole. See [Chapter 3](#) for more information about Console.

These steps are described in the next sections.

## Local Monitoring and Management

Under previous releases of the Java SE platform, to allow the JMX client access to a local Java VM, you had to set the following system property when you started the Java VM or Java application.

```
com.sun.management.jmxremote
```

Setting this property registered the Java VM platform's MBeans and published the Remote Method Invocation (RMI) connector via a private interface to allow JMX client applications to monitor a local Java platform, that is, a Java VM running on the same machine as the JMX client.

In the Java SE 6 platform, it is no longer necessary to set this system property. Any application that is started on the Java SE 6 platform will support the Attach API, and so will automatically be made available for local monitoring and management when needed.

For example, previously, to enable the JMX agent for the Java SE sample application Notepad, you would have to run the following commands.

```
% cd JDK_HOME/demo/jfc/Notepad
% java -Dcom.sun.management.jmxremote -jar Notepad.jar
```

In the above command, *JDK\_HOME* is the directory in which the Java Development Kit (JDK) is installed. In the Java SE 6 platform, you would simply have to run the following command to start Notepad.

```
% java -jar Notepad.jar
```

Once Notepad has been started, a JMX client using the Attach API can then enable the out-of-the-box management agent to monitor and manage the Notepad application.

**Note** – On Windows platforms, for security reasons, local monitoring and management is only supported if your default temporary directory is on a file system that allows the setting of permissions on files and directories (for example, on a New Technology File System (NTFS) file system). It is not supported on a File Allocation Table (FAT) file system, which provides insufficient access controls.

---

## Local Monitoring and Management Using JConsole

Local monitoring with JConsole is useful for development and creating prototypes. Using JConsole locally is not recommended for production environments, because JConsole itself consumes significant system resources. Rather, you should use JConsole on a remote system to isolate it from the platform being monitored.

However, if you do wish to perform local monitoring using JConsole, you start the tool by typing `jconsole` in a command shell. When you start `jconsole` without any arguments, it will automatically detect all local Java applications, and display a dialog box that enables you to select the application you want to monitor. Both JConsole and the application must be executed by the same user, since the monitoring and monitoring system uses the operating system's file permissions.

**Note** – To run JConsole from the command line, you must add `JDK_HOME/bin` to your path. Alternatively, you can enter the full path when you type the command.

---

For more information, see [Chapter 3](#).

## Remote Monitoring and Management

To enable monitoring and management from remote systems, you must set the following system property when you start the Java VM.

```
com.sun.management.jmxremote.port=portNum
```

In the property above, *portNum* is the port number through which you want to enable JMX RMI connections. Be sure to specify an unused port number. In addition to publishing an RMI connector for local access, setting this property publishes an additional RMI connector in a private read-only registry at the specified port using a well known name, "jmxrmi".

**Note** – You must set the above system property in addition to any properties you might set for security, as described in “[Using Password Authentication](#)” on page 20 and the sections that follow it.

---

Remote monitoring and management requires security, to ensure that unauthorized persons cannot control or monitor your application. Password authentication over the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) is enabled by default. You can disable password authentication and SSL separately, as described in the next sections.

After you have enabled the JMX agent for remote use, you can monitor your application using JConsole, as described in [“Remote Monitoring with JConsole” on page 23](#). How to connect to the management agent programmatically is described in [“Connecting to the JMX Agent Programmatically” on page 29](#).

## Using Password Authentication

By default, when you enable the JMX agent for remote monitoring, it uses password authentication. However, the way you set it up depends on whether you are in a single-user environment or a multiple-user environment.

Since passwords are stored in clear-text in the password file, it is not advisable to use your regular user name and password for monitoring. Instead, use the user names specified in the password file such as `monitorRole` and `controlRole`. For more information, see [“Using Password and Access Files” on page 24](#).

### ▼ To Set up a Single-User Environment

You set up the password file in the `JRE_HOME/lib/management` directory as follows.

- 1 **Copy the password template file, `jmxremote.password.template`, to `jmxremote.password`.**
- 2 **Set file permissions so that only the owner can read and write the password file.**
- 3 **Add passwords for roles such as `monitorRole` and `controlRole`.**

### ▼ To Set up a Multiple-User Environment

You set up the password file in the `JRE_HOME/lib/management` directory as follows.

- 1 **Copy the password template file, `jmxremote.password.template`, to your home directory and rename it to `jmxremote.password`.**
- 2 **Set file permissions so that only you can read and write the password file.**
- 3 **Add passwords for the roles such as `monitorRole` and `controlRole`.**
- 4 **Set the following system property when you start the Java VM.**

```
com.sun.management.jmxremote.password.file=pwFilePath
```

In the above property, *pwFilePath* is the path to the password file.



**Caution** – A potential security issue has been identified with password authentication for remote connectors when the client obtains the remote connector from an insecure RMI registry (the default). If an attacker starts a bogus RMI registry on the target server before the legitimate registry is started, then the attacker can steal clients' passwords. This scenario includes the case where you launch a Java VM with remote management enabled, using the system property `com.sun.management.jmxremote.port=portNum`, even when SSL is enabled. Although such attacks are likely to be noticed, it is nevertheless a vulnerability.

To avoid this problem, use SSL client certificates for authentication instead of passwords, or ensure that the client obtains the remote connector object securely, for example through a secure LDAP server or a file in a shared secure filesystem.

## Disabling Password Authentication

Password authentication for remote monitoring is enabled by default. To disable it, set the following system property when you start the Java VM.

```
com.sun.management.jmxremote.authenticate=false
```



**Caution** – This configuration is insecure. Any remote user who knows (or guesses) your JMX port number and host name will be able to monitor and control your Java application and platform. While it may be acceptable for development, it is not recommended for production systems.

When you disable password authentication, you can also disable SSL, as described in [“Disabling Security” on page 23](#). You may also want to disable passwords, but use SSL client authentication, as described in [“Enabling SSL Client Authentication” on page 22](#).

## Using SSL

SSL is enabled by default when you enable remote monitoring and management. To use SSL, you need to set up a digital certificate on the system where the JMX agent (the MBean server) is running and then configure SSL properly. You use the command-line utility `keytool` to work with certificates. The general procedure is as follows.

### ▼ To Set up SSL

#### 1 If you do not already have a key pair and certificate set up on the server:

- Generate a key pair with the `keytool -genkey` command.
- Request a signed certificate from a certificate authority (CA) with the `keytool -certreq` command.
- Import the certificate into your keystore with the `keytool -import` command. See [Importing Certificates](#) in the `keytool` documentation.

For more information and examples, see [keytool - Key and Certificate Management Tool \(Solaris and Linux\)](#) or [\(Windows platforms\)](#).

## 2 Configure SSL on the server system.

A full explanation of configuring and customizing SSL is beyond the scope of this document, but you generally need to set the system properties described in the list below.

System Property	Description
<code>javax.net.ssl.keyStore</code>	Keystore location.
<code>javax.net.ssl.keyStoreType</code>	Default keystore type.
<code>javax.net.ssl.keyStorePassword</code>	Default keystore password.
<code>javax.net.ssl.trustStore</code>	Truststore location.
<code>javax.net.ssl.trustStoreType</code>	Default truststore type.
<code>javax.net.ssl.trustStorePassword</code>	Default truststore password.

For more information about setting system properties, see “[Setting System Properties](#)” on page 17 above, or consult the following documents.

- [keytool - Key and Certificate Management Tool \(Solaris and Linux platforms\)](#)
- [keytool - Key and Certificate Management Tool \(Windows platforms\)](#)
- The section [Customizing the Default Key and Trust Stores, Store Types, and Store Passwords](#) in the JSSE Guide.

## Enabling RMI Registry Authentication

When setting up connections for monitoring remote applications, you can optionally bind the RMI connector stub to an RMI registry that is protected by SSL. This allows clients with the appropriate SSL certificates to get the connector stub that is registered in the RMI registry. To protect the RMI registry using SSL, you must set the following system property.

```
com.sun.management.jmxremote.registry.ssl=true
```

When this property is set to `true`, an RMI registry protected by SSL will be created and configured by the out-of-the-box management agent when the Java VM is started. The default value of this property is `false`. If this property is set to `true`, in order to have full security then SSL client authentication must also be enabled, as described in the next section.

## Enabling SSL Client Authentication

To enable SSL client authentication, set the following system property when you start the Java VM.

```
com.sun.management.jmxremote.ssl.need.client.auth=true
```

SSL must be enabled (the default), to use client SSL authentication. This configuration requires the client system to have a valid digital certificate. You must install a certificate and configure SSL on the client system, as described in [“Using SSL” on page 21](#). As stated in the previous section, if RMI registry SSL protection is enabled, then client SSL authentication must be set to `true`.

## Disabling SSL

To disable SSL when monitoring remotely, you must set the following system property when you start the Java VM.

```
com.sun.management.jmxremote.ssl=false
```

Password authentication will still be required unless you disable it, as specified in [“Disabling Password Authentication” on page 21](#).

## Disabling Security

To disable both password authentication and SSL (namely to disable *all* security), you should set the following system properties when you start the Java VM.

```
com.sun.management.jmxremote.authenticate=false  
com.sun.management.jmxremote.ssl=false
```



**Caution** – This configuration is insecure: any remote user who knows (or guesses) your port number and host name will be able to monitor and control your Java applications and platform. Furthermore, possible harm is not limited to the operations you define in your MBeans. A remote client could create a `javax.management.loading.MLet` MBean and use it to create new MBeans from arbitrary URLs, at least if there is no security manager. In other words, a rogue remote client could make your Java application execute arbitrary code.

Consequently, while disabling security might be acceptable for development, it is strongly recommended that you **do not disable security for production systems**.

## Remote Monitoring with JConsole

You can remotely monitor an application using JConsole, with or without security enabled.

### Remote Monitoring with JConsole with SSL Disabled

To monitor a remote application with SSL disabled, you would start JConsole with the following command.

```
% jconsole hostName:portNum
```

You can also omit the host name and port number, and enter them in the dialog box that JConsole provides.

## Remote Monitoring with JConsole with SSL Enabled

To monitor a remote application with SSL enabled, you need to set up the truststore on the system where JConsole is running and configure SSL properly. For example, you can create a keystore as described in the [JSSE Guide](#) and start your application (called `Server` in this example) with the following commands.

```
% java -Djavax.net.ssl.keyStore=keystore \  
-Djavax.net.ssl.keyStorePassword=password Server
```

If you created the keystore and started `Server` as shown above, then you would have to start JConsole as follows.

```
% jconsole -J-Djavax.net.ssl.trustStore=truststore \  
-J-Djavax.net.ssl.trustStorePassword=trustword
```

The above configuration authenticates the server only. If SSL client authentication is set up, you will need to provide a similar keystore for JConsole's keys, and an appropriate truststore for the application.

See [Customizing the Default Key and Trust Stores, Store Types, and Store Passwords](#) in the JSSE Guide for information.

For more information on using JConsole, see [Chapter 3](#).

# Using Password and Access Files

The password and access files control security for remote monitoring and management. These files are located by default in `JRE_HOME/lib/management` and are in the standard Java properties file format. For more information on the format, see the API reference for the `java.util.Properties` package.

## Password Files

The password file defines the different roles and their passwords. The access control file (`jmxremote.access` by default) defines the permitted access for each role. To be functional, a role must have an entry in both the password and the access files.

The JRE implementation contains a password file template named `jmxremote.password.template`. Copy this file to `JRE_HOME/lib/management/jmxremote.password` or to your home directory, and add the passwords for the roles defined in the access file.



You must ensure that only the owner has read and write permissions on this file, since it contains the passwords in clear text. For security reasons, the system checks that the file is only readable by the owner and exits with an error if it is not. Thus in a multiple-user environment, you should store the password file in private location such as your home directory.

Property names are roles, and the associated value is the role's password. For example, the following are sample entries in the password file.

#### EXAMPLE 2-1 An Example Password File

```
# The "monitorRole" role has password "QED".
# The "controlRole" role has password "R&D".
monitorRole QED
controlRole R&D
```

On Solaris and Linux systems, you can set the file permissions for the password file by running the following command.

```
chmod 600 jmxremote.password
```

For instructions on how to set file permissions on Windows platforms, see [Appendix A](#).

## Access Files

By default, the access file is named `jmxremote.access`. Property names are identities from the same space as the password file. The associated value must be either `readonly` or `readwrite`.

The access file defines roles and their access levels. By default, the access file defines the two following primary roles.

- `monitorRole`, which grants read-only access for monitoring.
- `controlRole`, which grants read-write access for monitoring and management.

An access control entry consists of a role name and an associated access level. The role name cannot contain spaces or tabs and must correspond to an entry in the password file. The access level can be either one of the following.

- `readonly`, which grants access to read an MBean's attributes. For monitoring, this means that a remote client in this role can read measurements but cannot perform any action that changes the environment of the running program. The remote client can also listen to MBean notifications.
- `readwrite`, which grants access to read and write an MBean's attributes, to invoke operations on them, and to create or remove them. This access should be granted to only trusted clients, since they can potentially interfere with the operation of an application.

A role should have only one entry in the access file. If a role has no entry, it has no access. If a role has multiple entries, then the last entry takes precedence. Typical predefined roles in the access file resemble the following.

**EXAMPLE 2-2** An Example Access File

```
# The "monitorRole" role has readonly access.
# The "controlRole" role has readwrite access.
monitorRole readonly
controlRole readwrite
```

## Out-of-the-Box Monitoring and Management Properties

You can set out-of-the-box monitoring and management properties in a configuration file or on the command line. Properties specified on the command line override properties in a configuration file. The default location for the configuration file is `JRE_HOME/lib/management/management.properties`. The Java VM reads this file if either of the command-line properties `com.sun.management.jmxremote` or `com.sun.management.jmxremote.port` are set. Management via the Simple Network Management Protocol (SNMP) uses the same configuration file. For more information about SNMP monitoring, see [Chapter 5](#).

You can specify a different location for the configuration file with the following command-line option.

```
com.sun.management.config.file=ConfigFilePath
```

In the property above, *ConfigFilePath* is the path to the configuration file.

[Table 2-1](#) describes all the out-of-the-box monitoring and management properties.

**TABLE 2-1** Out-of-the-Box Monitoring and Management Properties

Property	Description	Values
<code>com.sun.management.jmxremote</code>	Enables the JMX remote agent and local monitoring via a JMX connector published on a private interface used by JConsole and any other local JMX clients that use the Attach API. JConsole can use this connector if it is started by the same user as the user that started the agent. No password or access files are checked for requests coming via this connector.	<code>true / false</code> . Default is <code>true</code> .

TABLE 2-1 Out-of-the-Box Monitoring and Management Properties (Continued)

Property	Description	Values
<code>com.sun.management.jmxremote.port</code>	Enables the JMX remote agent and creates a remote JMX connector to listen through the specified port. By default, the SSL, password, and access file properties are used for this connector. It also enables local monitoring as described for the <code>com.sun.management.jmxremote</code> property.	Port number. No default.
<code>com.sun.management.jmxremote.registry.ssl</code>	Binds the RMI connector stub to an RMI registry protected by SSL.	<code>true / false</code> . Default is <code>false</code> .
<code>com.sun.management.jmxremote.ssl</code>	Enables secure monitoring via SSL. If <code>false</code> , then SSL is not used.	<code>true / false</code> . Default is <code>true</code> .
<code>com.sun.management.jmxremote.ssl.enabled.protocols</code>	A comma-delimited list of SSL/TLS protocol versions to enable. Used in conjunction with <code>com.sun.management.jmxremote.ssl</code> .	Default SSL/TLS protocol version.
<code>com.sun.management.jmxremote.ssl.enabled.cipher.suites</code>	A comma-delimited list of SSL/TLS cipher suites to enable. Used in conjunction with <code>com.sun.management.jmxremote.ssl</code> .	Default SSL/TLS cipher suites.
<code>com.sun.management.jmxremote.ssl.need.client.auth</code>	If this property is <code>true</code> and the property <code>com.sun.management.jmxremote.ssl</code> is also <code>true</code> , then client authentication will be performed.	<code>true / false</code> . Default is <code>true</code> .
<code>com.sun.management.jmxremote.authenticate</code>	If this property is <code>false</code> then JMX does not use passwords or access files: all users are allowed all access.	<code>true / false</code> . Default is <code>true</code> .
<code>com.sun.management.jmxremote.password.file</code>	Specifies location for password file. If <code>com.sun.management.jmxremote.authenticate</code> is <code>false</code> , then this property and the password and access files are ignored. Otherwise, the password file must exist and be in the valid format. If the password file is empty or nonexistent, then no access is allowed.	<code>JRE_HOME/lib/management/jmxremote.password</code>

Property	Description	Values
<code>com.sun.management.jmxremote.access.file</code>	Specifies location for the access file. If <code>com.sun.management.jmxremote.authenticate</code> is false, then this property and the password and access files are ignored. Otherwise, the access file must exist and be in the valid format. If the access file is empty or nonexistent, then no access is allowed.	<code>JRE_HOME/lib/management/jmxremote.access</code>
<code>com.sun.management.jmxremote.login.config</code>	Specifies the name of a Java Authentication and Authorization Service (JAAS) login configuration entry to use when the JMX agent authenticates users. When using this property to override the default login configuration, the named configuration entry must be in a file that is loaded by JAAS. In addition, the login modules specified in the configuration should use the name and password callbacks to acquire the user's credentials. For more information, see the API documentation for <code>javax.security.auth.callback.NameCallback</code> and <code>javax.security.auth.callback.PasswordCallback</code> .	Default login configuration is a file-based password authentication.

## Configuration Errors

If any errors occur during start up of the MBean server, the RMI registry, or the connector, the Java VM will throw an exception and exit. Configuration errors include the following.

- Failure to bind to the port number.
- Invalid password file.
- Invalid access file.
- Password file is readable by users other than the owner.

If your application runs a security manager, then additional permissions are required in the security permissions file.

## Connecting to the JMX Agent Programmatically

Once you have enabled the JMX agent, a client can use the following URL to access the monitoring service.

```
service:jmx:rmi:///jndi/rmi://hostName:portNum/jmxrmi
```

A client can create a connector for the agent by instantiating a `javax.management.remote.JMXServiceURL` object using the URL, and then creating a connection using the `JMXConnectorFactory.connect` method, shown in [Example 2-3](#).

**EXAMPLE 2-3** Creating a Connection Using `JMXConnectorFactory.connect`

```
JMXServiceURL u = new JMXServiceURL(
    "service:jmx:rmi:///jndi/rmi://" + hostName + ":" + portNum + "/jmxrmi");
JMXConnector c = JMXConnectorFactory.connect(u);
```

## Setting up Monitoring and Management Programmatically

As stated previously, in the Java SE platform version 6, you can create a JMX client that uses the [Attach API](#) to enable out-of-the-box monitoring and management of any applications that are started on the Java SE 6 platform, without having to configure the applications for monitoring when you launch them. The Attach API provides a way for tools to attach to and start agents in the target application. Once an agent is running, JMX clients (and other tools) are able to obtain the JMX connector address for that agent via a property list that is maintained by the Java VM on behalf of the agents. The properties in the list are accessible from tools that use the Attach API. So, if an agent is started in an application, and if the agent creates a property to represent a piece of configuration information, then that configuration information is available to tools that attach to the application.

The JMX agent creates a property with the address of the local JMX connector server. This allows JMX tools to attach to and get the connector address of an agent, if it is running.

[Example 2-4](#) shows code that could be used in a JMX tool to attach to a target VM, get the connector address of the JMX agent and connect to it.

**EXAMPLE 2-4** Attaching a JMX tool to a connector and getting the agent's address

```
static final String CONNECTOR_ADDRESS =
    "com.sun.management.jmxremote.localConnectorAddress";

// attach to the target application
VirtualMachine vm = VirtualMachine.attach(id);

// get the connector address
```

**EXAMPLE 2-4** Attaching a JMX tool to a connector and getting the agent's address *(Continued)*

```
String connectorAddress =
    vm.getAgentProperties().getProperty(CONNECTOR_ADDRESS);

// no connector address, so we start the JMX agent
if (connectorAddress == null) {
    String agent = vm.getSystemProperties().getProperty("java.home") +
        File.separator + "lib" + File.separator + "management-agent.jar";
    vm.loadAgent(agent);

    // agent is started, get the connector address
    connectorAddress =
        vm.getAgentProperties().getProperty(CONNECTOR_ADDRESS);
}

// establish connection to connector server
JMXServiceURL url = new JMXServiceURL(connectorAddress);
JMXConnector = JMXConnectorFactory.connect(url);
```

**Example 2-4** uses the `com.sun.tools.attach.VirtualMachine` class's `attach()` method to attach to a given Java VM so that it can read the properties that the target Java VM maintains on behalf of any agents running in it. If an agent is already running, then the `VirtualMachine` class's `getAgentProperties()` method is called to obtain the agent's address. The `getAgentProperties()` method returns a string property for the local connector address `com.sun.management.jmxremote.localConnectorAddress`, that you can use to connect to the local JMX agent.

If no agent is running already, then one is loaded by the `VirtualMachine` from `JRE_HOME/lib/management-agent.jar`, and its connector address is obtained by `getAgentProperties()`.

A connection to the agent is then established by calling `JMXConnectorFactory.connect` on a JMX service URL that has been constructed from this connector address.

## Mimicking Out-of-the-Box Management Using the JMX Remote API

As explained above, remote access to the out-of-the-box management agent is protected by authentication and authorization, and by SSL encryption, and all configuration is performed by setting system properties or by defining a `management.properties` file. In most cases, using the out-of-the-box management agent and configuring it through the `management.properties` file is more than sufficient to provide secure management of remote Java VMs. However, in some cases greater levels of security are required and in other cases certain system configurations do not allow the use of a `management.properties` file. Such cases might involve exporting the RMI server's

remote objects over a certain port to allow passage through a firewall, or exporting the RMI server's remote objects using a specific network interface in multi-homed systems. For such cases, the behavior of the out-of-the-box management agent can be mimicked by using the JMX Remote API directly to create, configure and deploy the management agent programmatically.

## Example of Mimicking Out-of-the-Box Management

This section provides an example of how to implement a JMX agent that identically mimics an out-of-the-box management agent. In exactly the same way as the out-of-the-box management agent, the agent created in [Example 2–5](#) will run on port 3000, will have a password file named `password.properties`, an access file named `access.properties` and it will implement the default configuration for SSL/TLS-based RMI Socket Factories, requiring server authentication only. This example assumes a keystore has already been created, as described in [“Using SSL” on page 21](#). Information about how to set up the SSL configuration can be found in the [JSSE Reference Guide](#).

To enable monitoring and management on an application named `com.example.MyApp` using the out-of-the-box JMX agent with the configuration described above, you would run `com.example.MyApp` with the following command.

```
% java -Dcom.sun.management.jmxremote.port=3000 \
-Dcom.sun.management.jmxremote.password.file=password.properties \
-Dcom.sun.management.jmxremote.access.file=access.properties \
-Djavax.net.ssl.keyStore=keystore \
-Djavax.net.ssl.keyStorePassword=password \
com.example.MyApp
```

---

**Note** – The `com.sun.management.jmxremote.*` properties could have been specified in a `management.properties` file instead of passing them at the command line. In that case, the system property `-Dcom.sun.management.config.file=management.properties` would be required to specify the location of the `management.properties` file.

---

[Example 2–5](#) shows the code you would need to write to create programmatically a JMX agent that will allow exactly the same monitoring and management on `com.example.MyApp` as would be possible using the command above.

### EXAMPLE 2-5 Mimicking an Out-of-the-Box JMX Agent Programmatically

```
package com.example;

import java.lang.management.*;
import java.rmi.registry.*;
import java.util.*;
import javax.management.*;
import javax.management.remote.*;
import javax.management.remote.rmi.*;
```

**EXAMPLE 2-5** Mimicking an Out-of-the-Box JMX Agent Programmatically *(Continued)*

```
import javax.rmi.ssl.*;

public class MyApp {

    public static void main(String[] args) throws Exception {

        // Ensure cryptographically strong random number generator used
        // to choose the object number - see java.rmi.server.ObjID
        //
        System.setProperty("java.rmi.server.randomIDs", "true");

        // Start an RMI registry on port 3000.
        //
        System.out.println("Create RMI registry on port 3000");
        LocateRegistry.createRegistry(3000);

        // Retrieve the PlatformMBeanServer.
        //
        System.out.println("Get the platform's MBean server");
        MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

        // Environment map.
        //
        System.out.println("Initialize the environment map");
        HashMap<String, Object> env = new HashMap<String, Object>();

        // Provide SSL-based RMI socket factories.
        //
        // The protocol and cipher suites to be enabled will be the ones
        // defined by the default JSSE implementation and only server
        // authentication will be required.
        //
        SslRMIClientSocketFactory csf = new SslRMIClientSocketFactory();
        SslRMIServerSocketFactory ssf = new SslRMIServerSocketFactory();
        env.put(RMIConnectorServer.RMI_CLIENT_SOCKET_FACTORY_ATTRIBUTE, csf);
        env.put(RMIConnectorServer.RMI_SERVER_SOCKET_FACTORY_ATTRIBUTE, ssf);

        // Provide the password file used by the connector server to
        // perform user authentication. The password file is a properties
        // based text file specifying username/password pairs.
        //
        env.put("jmx.remote.x.password.file", "password.properties");

        // Provide the access level file used by the connector server to
        // perform user authorization. The access level file is a properties
        // based text file specifying username/access level pairs where
```



**EXAMPLE 2-5** Mimicking an Out-of-the-Box JMX Agent Programmatically *(Continued)*

```

    // access level is either "readonly" or "readwrite" access to the
    // MBeanServer operations.
    //
    env.put("jmx.remote.x.access.file", "access.properties");

    // Create an RMI connector server.
    //
    // As specified in the JMXServiceURL the RMIServer stub will be
    // registered in the RMI registry running in the local host on
    // port 3000 with the name "jmxrmi". This is the same name the
    // out-of-the-box management agent uses to register the RMIServer
    // stub too.
    //
    System.out.println("Create an RMI connector server");
    JMXServiceURL url =
        new JMXServiceURL("service:jmx:rmi:///jndi/rmi://:3000/jmxrmi");
    JMXConnectorServer cs =
        JMXConnectorServerFactory.newJMXConnectorServer(url, env, mbs);

    // Start the RMI connector server.
    //
    System.out.println("Start the RMI connector server");
    cs.start();
}
}

```

Start this application with the following command.

```

java -Djavax.net.ssl.keyStore=keystore \
    -Djavax.net.ssl.keyStorePassword=password \
    com.example.MyApp

```

The `com.example.MyApp` application will enable the JMX agent and will be able to be monitored and managed in exactly the same way as if the Java platform's out-of-the-box management agent had been used. However, there is one slight but important difference between the RMI registry used by the out-of-the-box management agent and the one used by a management agent that mimics it. The RMI registry used by the out-of-the-box management agent is read-only, namely a single entry can be bound to it and once bound this entry cannot be unbound. This is not true of the RMI registry created in [Example 2-5](#).

Furthermore, both RMI registries are insecure as they do not use SSL/TLS. The RMI registries should be created using SSL/TLS-based RMI socket factories which require client authentication. This will prevent a client from sending its credentials to a rogue RMI server and will also prevent the RMI registry from giving access to the RMI server stub to a non-trusted client.

RMI registries which implement SSL/TLS RMI socket factories can be created by adding the following properties to your `management.properties` file.

```
com.sun.management.jmxremote.registry.ssl=true
com.sun.management.jmxremote.ssl.need.client.auth=true
```

**Example 2–5** mimics the main behavior of the out-of-the-box JMX agent, but does not replicate all the existing properties in the `management.properties` file. However, you could add other properties by modifying `com.example.MyApp` appropriately.

## Monitoring Applications through a Firewall

As stated above, the code in **Example 2–5** can be used to monitor applications through a firewall, which might not be possible if you use the out-of-the-box monitoring solution. The `com.sun.management.jmxremote.port` management property specifies the port where the RMI Registry can be reached but the ports where the `RMIServer` and `RMIConnection` remote objects are exported is chosen by the RMI stack. To export the remote objects (`RMIServer` and `RMIConnection`) to a given port you need to create your own RMI connector server programmatically, as described in **Example 2–5**. However, you must specify the `JMXServiceURL` as follows:

```
JMXServiceURL url = new JMXServiceURL("service:jmx:rmi://localhost:" +
    port1 + "/jndi/rmi://localhost:" + port2 + "/jmxrmi");
```

In the URL above, `port1` is the port number on which the `RMIServer` and `RMIConnection` remote objects are exported and `port2` is the port number of the RMI Registry.

## Using an Agent Class to Instrument an Application

The Java SE platform provides services that allow Java programming language agents to instrument programs running on the Java VM. Creating an instrumentation agent means you do not have to add any new code to your application in order to allow it to be monitored. Instead of implementing monitoring and management in your application's static `main` method you implement it in a separate agent class, and start your application with the `-javaagent` option specified. See the API reference documentation for the [java.lang.instrument](#) package for full details about how to create an agent class to instrument your applications.

The following procedure shows how you can adapt the code of `com.example.MyApp` to make an agent to instrument any other application for monitoring and management.

### ▼ Creating an Agent Class to Instrument an Application

#### 1 Create a `com.example.MyAgent` class.

Create a class called `com.example.MyAgent`, declaring a `premain` method rather than a `main` method.

```
package com.example;
```

```
[...]
```

```
public class MyAgent {

    public static void premain(String args) throws Exception {

        [...]
    }
}
```

The rest of the code for the `com.example.MyAgent` class can be exactly the same as the `com.example.MyApp` class shown in [Example 2-5](#).

**2 Compile the `com.example.MyAgent` class.**

**3 Create a manifest file, `MANIFEST.MF`, with a `Premain-Class` entry.**

An agent is deployed as a Java archive (JAR) file. An attribute in the JAR file manifest specifies the agent class which will be loaded to start the agent. Create a file called `MANIFEST.MF`, containing the following line.

```
Premain-Class: com.example.MyAgent
```

**4 Create a JAR file, `MyAgent.jar`.**

The JAR file should contain the following files.

- `META-INF/MANIFEST.MF`
- `com/example/MyAgent.class`

**5 Start an application, specifying the agent to provide monitoring and management services.**

You can use `com.example.MyAgent` to instrument any application for monitoring and management. This example uses the Notepad application.

```
% java -javaagent:MyAgent.jar -Djavax.net.ssl.keyStore=keystore \
    -Djavax.net.ssl.keyStorePassword=password -jar Notepad.jar
```

The `com.example.MyAgent` agent is specified using the `-javaagent` option when you start Notepad. Also, if your `com.example.MyAgent` application replicates the same code as the `com.example.MyApp` application shown in [Example 2-5](#), then you will need to provide the keystore and password because the RMI connector server is protected by SSL.



# Using JConsole

---

The JConsole graphical user interface is a monitoring tool that complies to the Java Management Extensions (JMX) specification. JConsole uses the extensive instrumentation of the Java Virtual Machine (Java VM) to provide information about the performance and resource consumption of applications running on the Java platform.

In the Java Platform, Standard Edition (Java SE platform) 6, JConsole has been updated to present the look and feel of the Windows and GNOME desktops (other platforms will present the standard Java graphical look and feel). The screen captures presented in this document were taken from an instance of the interface running on Windows XP.

## Starting JConsole

The `jconsole` executable can be found in `JDK_HOME/bin`, where `JDK_HOME` is the directory in which the Java Development Kit (JDK) is installed. If this directory is in your system path, you can start JConsole by simply typing `jconsole` in a command (shell) prompt. Otherwise, you have to type the full path to the executable file.

## Command Syntax

You can use JConsole to monitor both local applications, namely those running on the same system as JConsole, as well as remote applications, namely those running on other systems.

---

**Note** – Using JConsole to monitor a local application is useful for development and for creating prototypes, but is not recommended for production environments, because JConsole itself consumes significant system resources. Remote monitoring is recommended to isolate the JConsole application from the platform being monitored.

---

For a complete reference on the syntax of the `jconsole` command, see the manual page for the `jconsole` command: [Java Monitoring and Management Console](#).

## Setting up Local Monitoring

You start JConsole by typing the following command at the command line.

```
% jconsole
```

When JConsole starts, you will be given a choice of all the Java applications that are running locally that JConsole can connect to.

If you want to monitor a specific application, and you know that application's process ID, then you can also start JConsole so that it connects to that application. This application must be running with the same user ID as JConsole. The command syntax to start JConsole for local monitoring of a specific application is the following.

```
% jconsole processID
```

In the command above *processID* is the application's process ID (PID). You can determine an application's PID in the following ways:

- On UNIX or Linux systems, you can use the `ps` command to find the PID of the `java` instance that is running.
- On Windows systems, you can use the Task Manager to find the PID of `java` or `javaw`.
- You can also use the `jps` command-line utility to determine PIDs. See the manual page for the [Java Virtual Machine Process Status Tool](#).

For example, if you determined that the process ID of the Notepad application is 2956, then you would start JConsole with the following command.

```
% jconsole 2956
```

Both JConsole and the application must be executed by the same user. The management and monitoring system uses the operating system's file permissions. If you do not specify a process ID, JConsole will automatically detect all local Java applications, and display a dialog box that lets you select which one you want to monitor (see [“Connecting to a JMX Agent” on page 39](#)).

For more information, see [“Local Monitoring and Management” on page 18](#).

## Setting up Remote Monitoring

To start JConsole for remote monitoring, you use the following command syntax.

```
% jconsole hostName:portNum
```

In the command above, *hostName* is the name of the system running the application and *portNum* is the port number you specified when you enabled the JMX agent when you started the Java VM. For more information, see [“Remote Monitoring and Management” on page 19](#).

If you do not specify a host name/port number combination, then JConsole will display a connection dialog box ([“Connecting to a JMX Agent” on page 39](#)) to enable you to enter a host name and port number.

## Setting up Secure Remote Monitoring

You can also start JConsole so that monitoring will be performed over a connection that is secured using Secure Sockets Layer (SSL). The command to start JConsole with a secure connection is given in “Remote Monitoring with JConsole with SSL Enabled” on page 24 in Chapter 2.

## Connecting to a JMX Agent

If you start JConsole with arguments specifying a JMX agent to connect to, it will automatically start monitoring the specified Java VM. You can connect to a different host at any time by choosing Connection | New Connection and entering the necessary information.

Otherwise, if you do not provide any arguments when you start JConsole, the first thing you see is the connection dialog box. This dialog box has two options, allowing connections to either Local or Remote processes.

## Dynamic Attach

Under previous releases of the Java SE platform, applications that you wanted to monitor with JConsole needed to be started with the following option.

```
% -Dcom.sun.management.jmxremote
```

However, the version of JConsole provided with the Java SE 6 platform can attach to any application that supports the Attach API. In other words, any application that is started in the Java SE 6 HotSpot VM is detected automatically by JConsole, and does not need to be started using the above command-line option.

## Connecting JConsole to a Local Process

If you start JConsole without providing a specific JMX agent to connect to, you will see the following dialog window.

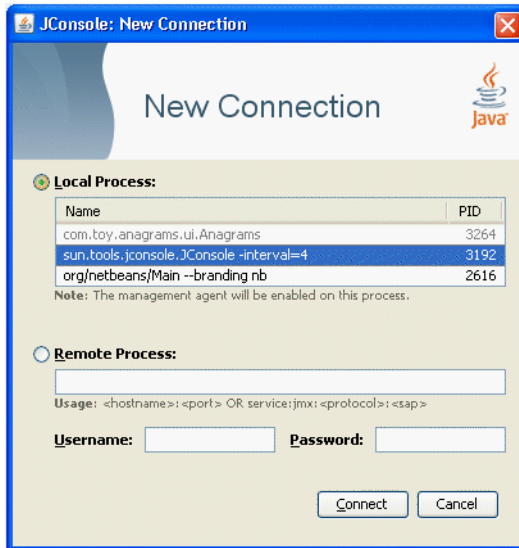


FIGURE 3-1 Creating a Connection to a Local Process

The Local Process option lists any Java VMs running on the local system that were started with the same user ID as JConsole, along with their process ID and their class and/or argument information. To connect JConsole to your application, select the application you want to monitor, then click the Connect button. The list of local processes includes applications running in the following types of Java VM.

- **Applications with the management agent enabled.** These include applications on the Java SE 6 platform or on the J2SE 5.0 platform that were started with the `-Dcom.sun.management.jmxremote` option or with the `-Dcom.sun.management.jmxremote.port` option specified. In addition, the list also includes any applications that were started on the Java SE 6 platform without any management properties but which are later attached to by JConsole, which enables the management agent at runtime.
- **Applications that are attachable, with the management agent disabled.** An *attachable* application supports loading the management agent at runtime. Attachable applications include applications that are started on the Java SE 6 platform that support the Attach API. Applications which support dynamic attach do not require the management agent to be started by specifying the `com.sun.management.jmxremote` or `com.sun.management.jmxremote.port` options at the command line, and JConsole does not need to connect to the management agent before the application is started. If you select this application, you will be informed in a note onscreen that the management agent will be enabled when the connection is made. In the example connection dialog shown in Figure 3-1, the NetBeans IDE and JConsole itself were both started within a Java SE 6 platform VM. Both appear in normal text, meaning that JConsole can connect to them. In Figure 3-1, JConsole is selected, and the note is visible.



- **Applications that are not attachable, with the management agent disabled.** These include applications started on a J2SE 1.4.2 platform or started on a J2SE 5.0 platform without the `-Dcom.sun.management.jmxremote` or `com.sun.management.jmxremote.port` options. These applications appear grayed-out in the table and JConsole cannot connect to them. In the example connection dialog shown in Figure 3–1, the Anagrams application was started with a J2SE 5.0 platform VM without any of the management properties to enable the JMX agent, and consequently shows up in gray and cannot be selected.

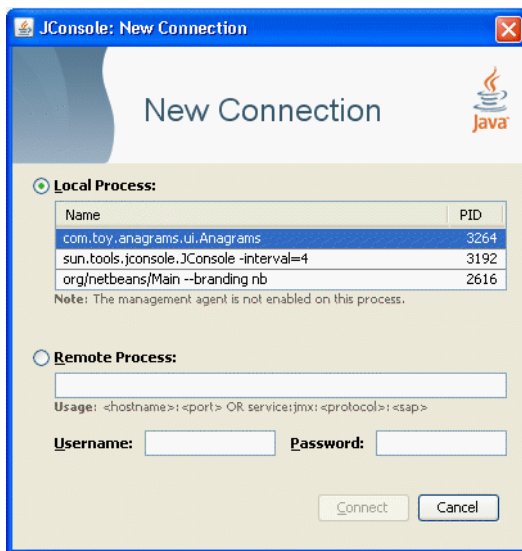


FIGURE 3–2 Attempting to Connect to an Application without the Management Agent Enabled

In the example connection dialog shown in Figure 3–2, you can see that the Anagrams application has been selected by clicking on it, but the Connect button remains grayed-out and a note has appeared informing you that the management agent is not enabled for this process. JConsole cannot connect to Anagrams because it was not started with the correct Java VM or with the correct options.

## Connecting JConsole to a Remote Process

When the connection dialog opens, you are also given the option of connecting to a remote process.

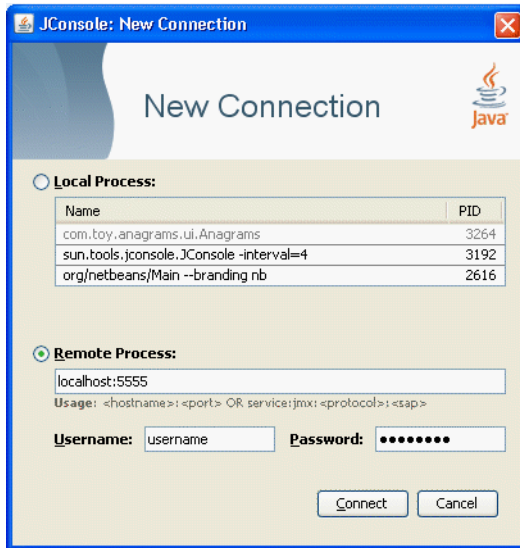


FIGURE 3-3 Creating a Connection to a Remote Process

To monitor a process running on a remote Java VM, you must provide the following information.

- Host name: name of the machine on which the Java VM is running.
- Port number: the JMX agent port number you specified when you started the Java VM.
- User name and password: the user name and password to use (required only if monitoring a Java VM through a JMX agent that requires password authentication).

For information about setting the port number of the JMX agent, see [“Enabling the Out-of-the-Box Management”](#) on page 18. For information about user names and passwords, see [“Using Password and Access Files”](#) on page 24.

To monitor the Java VM that is running JConsole, simply click **Connect**, using host `localhost` and the port `0`.

## Connecting Using a JMX Service URL

You can also use the Remote Process option to connect to other JMX agents by specifying their JMX service URL, and the user name and password. The syntax of a JMX service URL requires that you provide the transport protocol used to make the connection, as well as a service access point. The full syntax for a JMX service URL is described in the API documentation for `javax.management.remote.JMXServiceURL`.

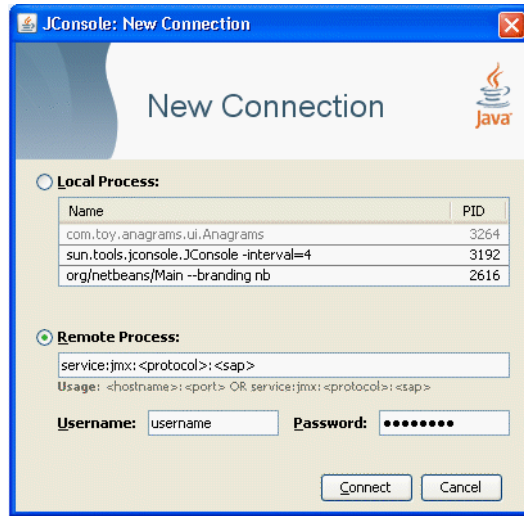


FIGURE 3-4 Connecting to a JMX Agent Using the JMX Service URL

If the JMX agent uses a connector which is not included in the Java platform, you need to add the connector classes to the class path when you run the `jconsole` command, as follows.

```
% jconsole -J-Djava.class.path=JAVA_HOME/lib/jconsole.jar:JAVA_HOME/lib/tools.jar:connector-path
```

In the command above, *connector-path* is the directory or the Java archive (Jar) file containing the connector classes that are not included in the JDK, that are to be used by JConsole.

## Presenting the JConsole Tabs

Once you have connected JConsole to an application, JConsole is composed of six tabs.

- **Overview:** Displays overview information about the Java VM and monitored values.
- **Memory:** Displays information about memory use.
- **Threads:** Displays information about thread use.
- **Classes:** Displays information about class loading.
- **VM:** Displays information about the Java VM.
- **MBeans:** Displays information about MBeans.

You can use the green connection status icon in the upper right-hand corner of JConsole at any time, to disconnect from or reconnect to a running Java VM. You can connect to any number of running Java VMs at a time by selecting `Connection` then `New Connection` from the drop-down menu.

## Viewing Overview Information

The Overview tab displays graphical monitoring information about CPU usage, memory usage, thread counts, and the classes loaded in the Java VM, all in a single screen.

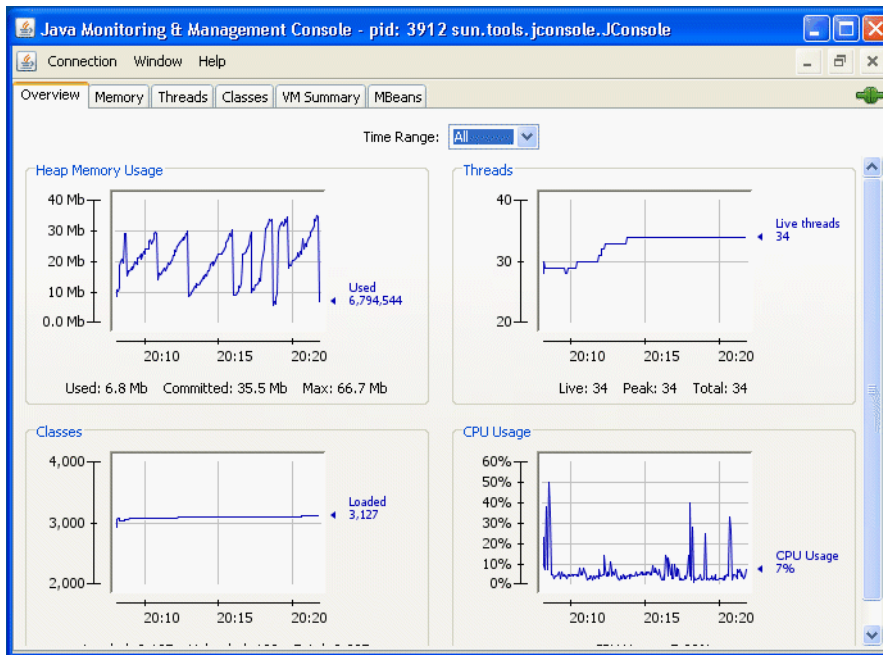


FIGURE 3-5 Overview Tab

The Overview tab provides an easy way to correlate information that was previously only available by switching between multiple tabs.

## Saving Chart Data

JConsole allows you to save the data presented in the charts in a Comma Separated Values (CSV) file. To save data from a chart, simply right-click on any chart, select *Save data as . . .*, and then specify the file in which the data will be saved. You can save the data from any of the charts displayed in any of JConsole's different tabs in this way.

The CSV format is commonly used for data exchange between spreadsheet applications. The CSV file can be imported into spreadsheet applications and can be used to create diagrams in these applications. The data is presented as two or more named columns, where the first column represents the time stamps. After importing the file into a spreadsheet application, you will usually need to select the first column and change its format to be "date" or "date/time" as appropriate.

## Monitoring Memory Consumption

The Memory tab provides information about memory consumption and memory pools.

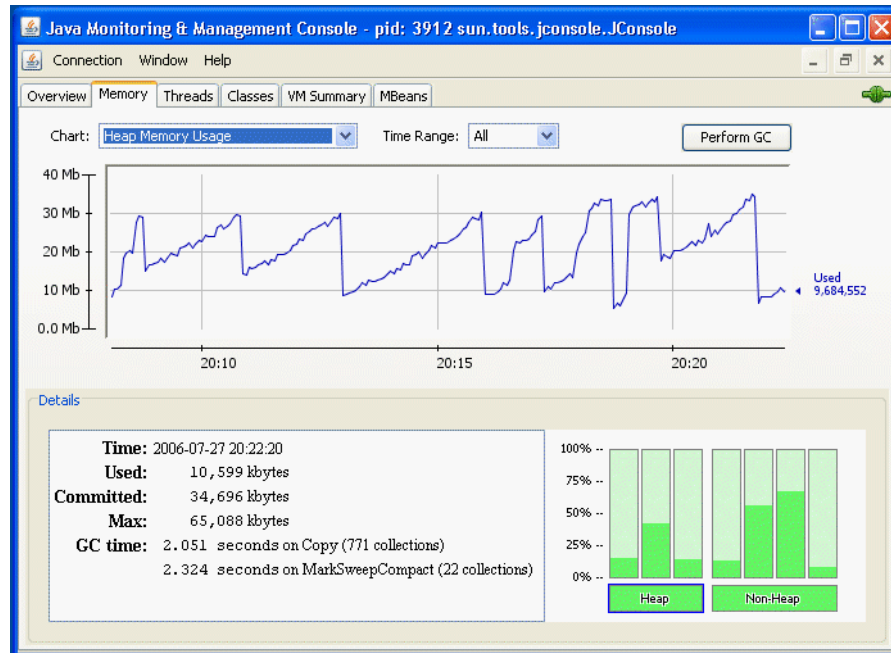


FIGURE 3-6 Memory Tab

The Memory tab features a “Perform GC” button that you can click to perform garbage collection whenever you want. The chart shows the memory use of the Java VM over time, for heap and non-heap memory, as well as for specific memory pools. The memory pools available depend on which version of the Java VM is being used. For the HotSpot Java VM, the memory pools for serial garbage collection are the following.

- *Eden Space (heap)*: The pool from which memory is initially allocated for most objects.
- *Survivor Space (heap)*: The pool containing objects that have survived the garbage collection of the Eden space.
- *Tenured Generation (heap)*: The pool containing objects that have existed for some time in the survivor space.
- *Permanent Generation (non-heap)*: The pool containing all the reflective data of the virtual machine itself, such as class and method objects. With Java VMs that use class data sharing, this generation is divided into read-only and read-write areas.
- *Code Cache (non-heap)*: The HotSpot Java VM also includes a code cache, containing memory that is used for compilation and storage of native code.

You can display different charts for charting the consumption of these memory pools by choosing from the options in the `Chart` drop-down menu. Also, clicking on either of the `Heap` or `Non-Heap` bar charts in the bottom right-hand corner will switch the chart displayed. Finally, you can specify the time range over which you track memory usage by selecting from the options in the `Time Range` drop-down menu.

For more information about these memory pools, see [“Garbage Collection” on page 47](#) below.

The **Details** area shows several current memory metrics:

- *Used*: the amount of memory currently used, including the memory occupied by all objects, both reachable and unreachable.
- *Committed*: the amount of memory guaranteed to be available for use by the Java VM. The amount of committed memory may change over time. The Java virtual machine may release memory to the system and the amount of committed memory could be less than the amount of memory initially allocated at start up. The amount of committed memory will always be greater than or equal to the amount of used memory.
- *Max*: the maximum amount of memory that can be used for memory management. Its value may change or be undefined. A memory allocation may fail if the Java VM attempts to increase the used memory to be greater than committed memory, even if the amount used is less than or equal to `max` (for example, when the system is low on virtual memory).
- *GC time*: the cumulative time spent on garbage collection and the total number of invocations. It may have multiple rows, each of which represents one garbage collector algorithm used in the Java VM.

The bar chart on the lower right-hand side shows the memory consumed by the memory pools in heap and non-heap memory. The bar will turn red when the memory used exceeds the memory usage threshold. You can set the memory usage threshold through an attribute of the `MemoryMXBean`.

## Heap and Non-Heap Memory

The Java VM manages two kinds of memory: heap and non-heap memory, both of which are created when the Java VM starts.

- *Heap memory* is the runtime data area from which the Java VM allocates memory for all class instances and arrays. The heap may be of a fixed or variable size. The garbage collector is an automatic memory management system that reclaims heap memory for objects.
- *Non-heap memory* includes a method area shared among all threads and memory required for the internal processing or optimization for the Java VM. It stores per-class structures such as a runtime constant pool, field and method data, and the code for methods and constructors. The method area is logically part of the heap but, depending on the implementation, a Java VM may not garbage collect or compact it. Like the heap memory, the method area may be of a fixed or variable size. The memory for the method area does not need to be contiguous.

In addition to the method area, a Java VM may require memory for internal processing or optimization which also belongs to non-heap memory. For example, the Just-In-Time (JIT) compiler requires memory for storing the native machine code translated from the Java VM code for high performance.

## Memory Pools and Memory Managers

Memory pools and memory managers are key aspects of the Java VM's memory system.

- A *memory pool* represents a memory area that the Java VM manages. The Java VM has at least one memory pool and it may create or remove memory pools during execution. A memory pool can belong either to heap or to non-heap memory.
- A *memory manager* manages one or more memory pools. The garbage collector is a type of memory manager responsible for reclaiming memory used by unreachable objects. A Java VM may have one or more memory managers. It may add or remove memory managers during execution. A memory pool can be managed by more than one memory manager.

## Garbage Collection

Garbage collection (GC) is how the Java VM frees memory occupied by objects that are no longer referenced. It is common to think of objects that have active references as being "alive" and non-referenced (or unreachable) objects as "dead." Garbage collection is the process of releasing memory used by the dead objects. The algorithms and parameters used by GC can have dramatic effects on performance.

The Java HotSpot VM garbage collector uses generational GC. Generational GC takes advantage of the observation that most programs conform to the following generalizations.

- They create many objects that have short lives, for example, iterators and local variables.
- They create some objects that have very long lives, for example, high level persistent objects.

Generational GC divides memory into several generations, and assigns one or more memory pools to each. When a generation uses up its allotted memory, the VM performs a partial GC (also called a minor collection) on that memory pool to reclaim memory used by dead objects. This partial GC is usually much faster than a full GC.

The Java HotSpot VM defines two generations: the young generation (sometimes called the "nursery") and the old generation. The young generation consists of an "Eden space" and two "survivor spaces." The VM initially assigns all objects to the Eden space, and most objects die there. When it performs a minor GC, the VM moves any remaining objects from the Eden space to one of the survivor spaces. The VM moves objects that live long enough in the survivor spaces to the "tenured" space in the old generation. When the tenured generation fills up, there is a full GC that is often much slower because it involves all live objects. The permanent generation holds all the reflective data of the virtual machine itself, such as class and method objects.

The default arrangement of generations looks something like [Figure 3–7](#).

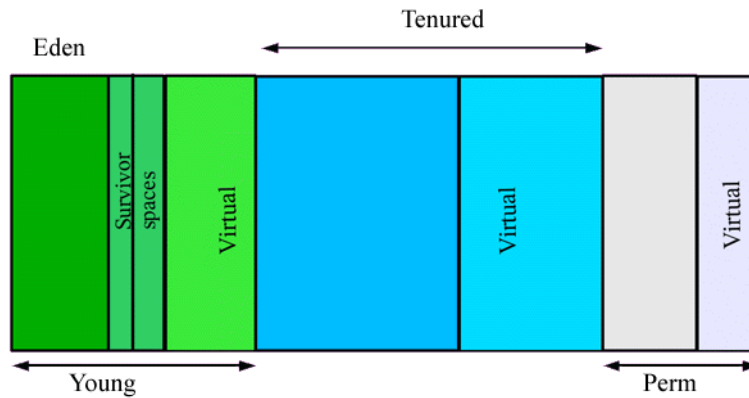


FIGURE 3-7 Generations of Data in Garbage Collection

If the garbage collector has become a bottleneck, you can improve performance by customizing the generation sizes. Using JConsole, you can investigate the sensitivity of your performance metric by experimenting with the garbage collector parameters. For more information, see [Tuning Garbage Collection with the 5.0 HotSpot VM](#).

## Monitoring Thread Use

The Threads tab provides information about thread use.



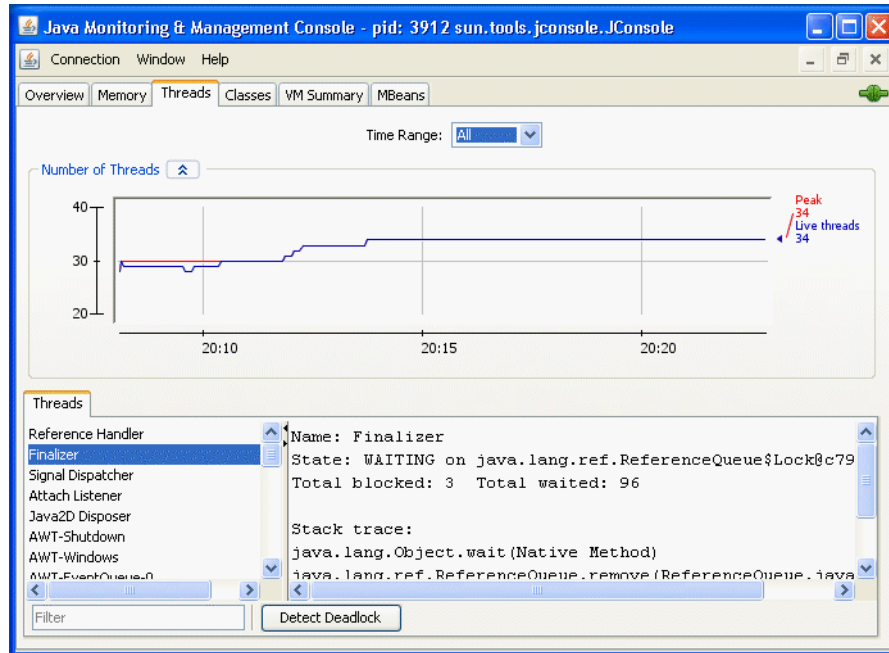


FIGURE 3-8 Threads Tab

The Threads list in the lower left corner lists all the active threads. If you enter a string in the Filter field, the Threads list will show only those threads whose name contains the string you enter. Click on the name of a thread in the Threads list to display information about that thread to the right, including the thread name, state, and stack trace.

The chart shows the number of live threads over time. Two lines are shown.

- *Red*: peak number of threads
- *Blue*: number of live threads.

The Threading MBean provides several other useful operations that are not covered by the Threads tab.

- `findMonitorDeadlockedThreads`: Detects if any threads are deadlocked on the object monitor locks. This operation returns an array of deadlocked thread IDs.
- `getThreadInfo`: Returns the thread information. This includes the name, stack trace, and the monitor lock that the thread is currently blocked on, if any, and which thread is holding that lock, as well as thread contention statistics.
- `getThreadCpuTime`: Returns the CPU time consumed by a given thread

You can access these additional features via the MBeans tab by selecting the Threading MBean in the MBeans tree. This MBean lists all the attributes and operations for accessing threading information in the Java VM being monitored. See “[Monitoring and Managing MBeans](#)” on page 53.

## Detecting Deadlocked Threads

To check if your application has run into a deadlock (for example, your application seems to be hanging), deadlocked threads can be detected by clicking on the "Detect Deadlock" button. If any deadlocked threads are detected, these are displayed in a new tab that appears next to the "Threads" tab, as shown in [Figure 3-9](#).

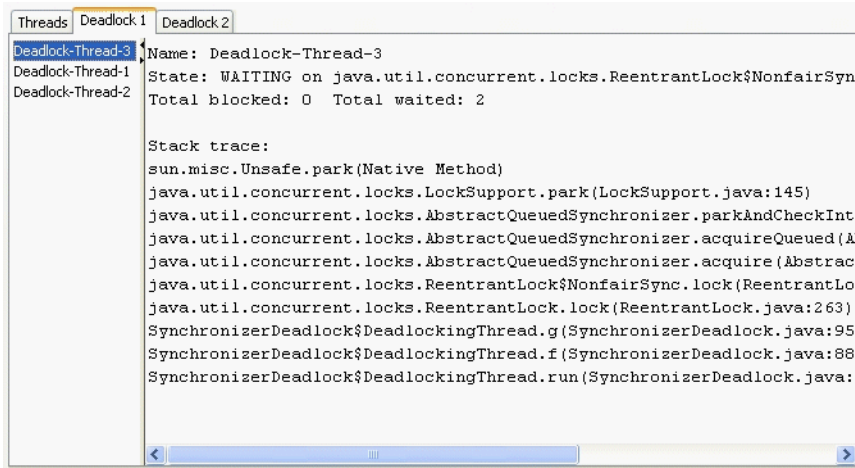


FIGURE 3-9 Deadlocked Threads

The Detect Deadlock button will detect deadlock cycles involving object monitors and `java.util.concurrent` ownable synchronizers (see the API specification documentation for [java.lang.management.LockInfo](#)). Monitoring support for `java.util.concurrent` locks has been added in Java SE 6. If JConsole connects to a J2SE 5.0 VM, the Detect Deadlock mechanism will only find deadlocks related to object monitors. JConsole will not show any deadlocks related to ownable synchronizers.

See the API documentation for `java.lang.Thread` for more information about threads and daemon threads.

## Monitoring Class Loading

The Classes tab displays information about class loading.

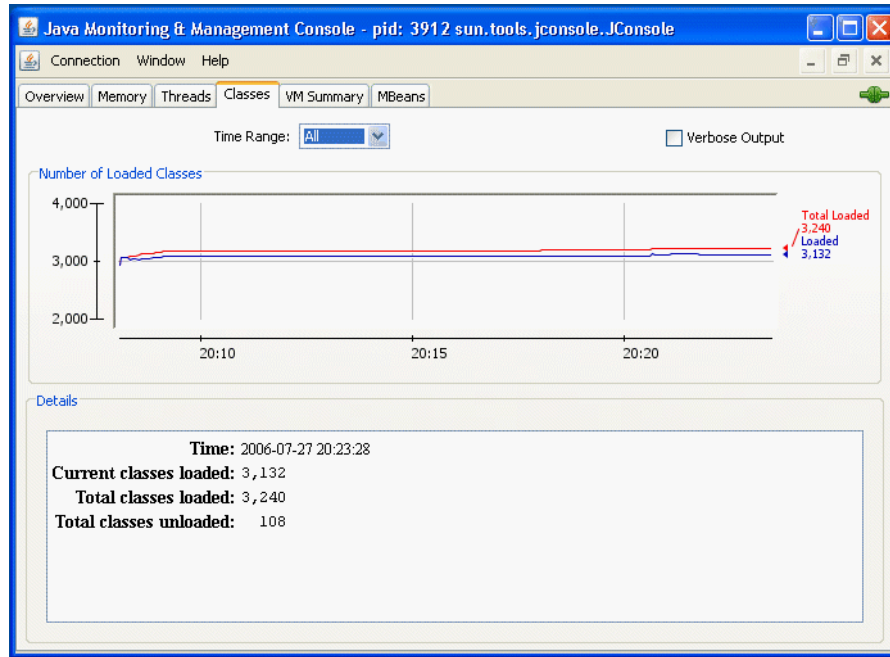


FIGURE 3-10 Classes Tab

The chart plots the number of classes loaded over time.

- The red line is the total number of classes loaded (including those subsequently unloaded).
- The blue line is the current number of classes loaded.

The Details section at the bottom of the tab displays the total number of classes loaded since the Java VM started, the number currently loaded and the number unloaded. You can set the tracing of class loading to verbose output by checking the checkbox in the top right-hand corner.

## Viewing VM Information

The VM Summary tab provides information about the Java VM.

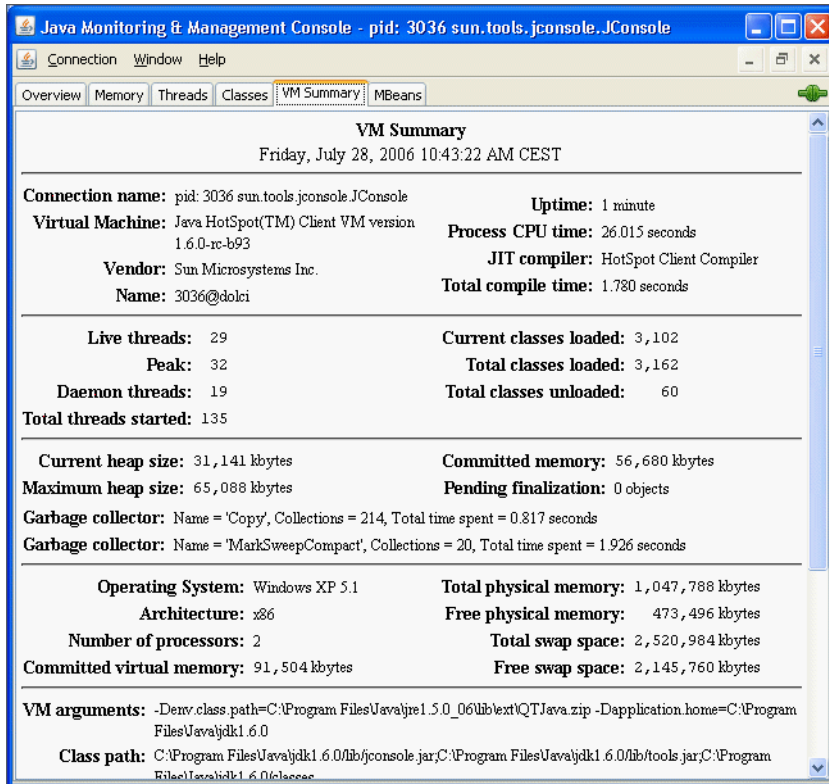


FIGURE 3-11 VM Summary Tab

The information presented in this tab includes the following.

- **Summary**
  - *Uptime:* Total amount of time since the Java VM was started.
  - *Process CPU Time:* Total amount of CPU time that the Java VM has consumed since it was started.
  - *Total Compile Time:* Total accumulated time spent in JIT compilation. The Java VM determines when JIT compilation occurs. The Hotspot VM uses adaptive compilation, in which the VM launches an application using a standard interpreter, but then analyzes the code as it runs to detect performance bottlenecks, or "hot spots".
- **Threads**
  - *Live threads:* Current number of live daemon threads plus non-daemon threads.
  - *Peak:* Highest number of live threads since Java VM started.
  - *Daemon threads:* Current number of live daemon threads.

- *Total threads started*: Total number of threads started since Java VM started, including daemon, non-daemon, and terminated threads.
- **Classes**
  - *Current classes loaded*: Number of classes currently loaded into memory.
  - *Total classes loaded*: Total number of classes loaded into memory since the Java VM started, including those that have subsequently been unloaded.
  - *Total classes unloaded*: Number of classes unloaded from memory since the Java VM started.
- **Memory**
  - *Current heap size*: Number of kilobytes currently occupied by the heap.
  - *Committed memory*: Total amount of memory allocated for use by the heap.
  - *Maximum heap size*: Maximum number of kilobytes occupied by the heap.
  - *Objects pending for finalization*: Number of objects pending for finalization.
  - *Garbage collector*: Information about garbage collection, including the garbage collector names, number of collections performed, and total time spent performing GC.
- **Operating System**
  - *Total physical memory*: Amount of random-access memory (RAM) the operating system has.
  - *Free physical memory*: Amount of free RAM available to the operating system.
  - *Committed virtual memory*: Amount of virtual memory guaranteed to be available to the running process.
- **Other Information**
  - *VM arguments*: The input arguments the application passed to the Java VM, not including the arguments to the main method.
  - *Class path*: The class path that is used by the system class loader to search for class files.
  - *Library path*: The list of paths to search when loading libraries.
  - *Boot class path*: The boot class path is used by the bootstrap class loader to search for class files.

## Monitoring and Managing MBeans

The MBeans tab displays information about all the MBeans registered with the platform MBean server in a generic way. The MBeans tab allows you to access the full set of the platform MXBean instrumentation, including that which is not visible in the other tabs. In addition, you can monitor and manage your application's MBeans using the MBeans tab.

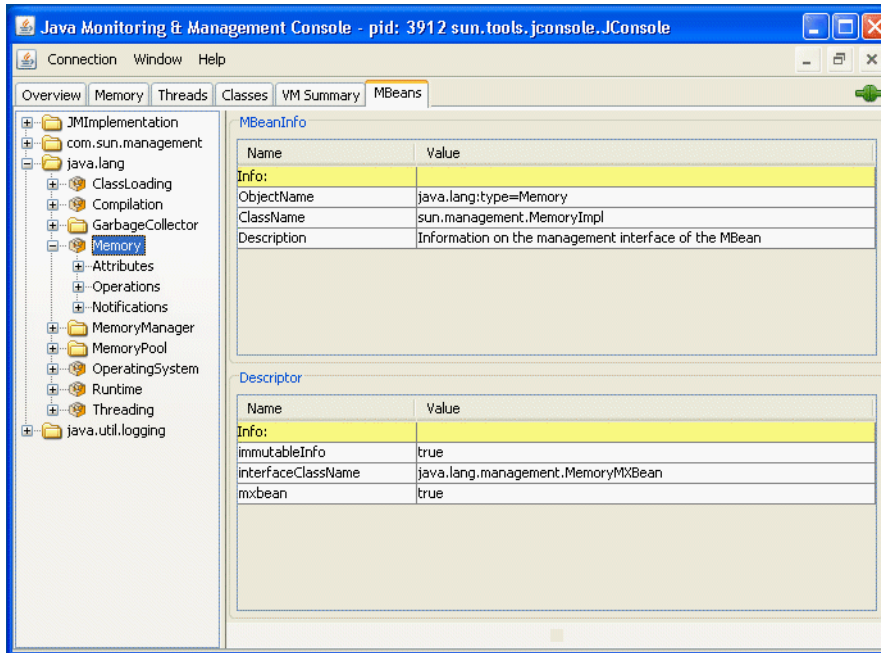


FIGURE 3-12 MBeans Tab

The tree on the left shows all the MBeans currently running. When you select an MBean in the tree, its `MBeanInfo` and its `MBean Descriptor` are both displayed on the right, and any attributes, operations or notifications appear in the tree below it.

All the platform MXBeans and their various operations and attributes are accessible via JConsole's MBeans tab.

## Constructing the MBean Tree

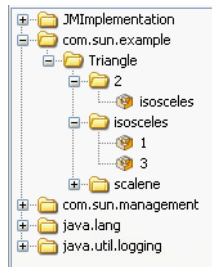
By default, the MBeans are displayed in the tree based on their object names. The order of key properties specified when the object names are created is preserved by JConsole when it adds MBeans to the MBean tree. The exact key property list that JConsole will use to build the MBean tree will be the one returned by the method `ObjectName.getKeyPropertyListString()`, with `type` as the first key, and `j2eeType`, if present, as the second key.

However, relying on the default order of the `ObjectName` key properties can sometimes lead to unexpected behavior when JConsole renders the MBean tree. For example, if two object names have similar keys but their key order differs, then the corresponding MBeans will not be created under the same node in the MBean tree.

For example, suppose you create `Triangle` MBean objects with the following names.

```
com.sun.example:type=Triangle,side=isosceles,name=1
com.sun.example:type=Triangle,name=2,side=isosceles
com.sun.example:type=Triangle,side=isosceles,name=3
```

As far as the JMX technology is concerned, these objects will be treated in exactly the same way. The order of the keys in the object name makes no difference to the JMX technology. However, if JConsole connects to these MBeans and the default MBean tree rendering is used, then the object `com.sun.example:type=Triangle,name=2,side=isosceles` will end up being created under the `Triangle` node, in a node called `2`, which in turn will contain a sub-node called `isosceles`. The other two isosceles triangles, `name=1` and `name=3`, will be grouped together under `Triangle` in a different node called `isosceles`, as shown in [Figure 3-13](#).



**FIGURE 3-13** Example of Unexpected MBean Tree Rendering

To avoid this problem, you can specify the order in which the MBeans are displayed in the tree by supplying an ordered key property list when you start JConsole at the command line. This is achieved by setting the system property `com.sun.tools.jconsole.mbeans.keyPropertyList`, as shown in the following command.

```
% jconsole -J-Dcom.sun.tools.jconsole.mbeans.keyPropertyList=key[,key]*
```

The key property list system property takes a comma-separated list of keys, in the order of your choosing, where *key* must be a string representing an object name key or an empty string. If a key specified in the list does not apply to a particular MBean, then that key will be discarded. If an MBean has more keys than the ones specified in the key property list, then the key order defined by the value returned by `ObjectName.getKeyPropertyListString()` will be used to complete the key order defined by `keyPropertyList`. Therefore, specifying an empty list of keys simply means that JConsole will display keys in the order they appear in the MBean's `ObjectName`.

So, returning to the example of the `Triangle` MBeans cited above, you could choose to start JConsole specifying the `keyPropertyList` system property, so that all your MBeans will be grouped according to their `side` key property first, and their `name` key property second. To do this, you would start JConsole with the following command.

```
% jconsole -J-Dcom.sun.tools.jconsole.mbeans.keyPropertyList=side,name
```

Starting JConsole with this system property specified would produce the MBean tree shown in [Figure 3-14](#).

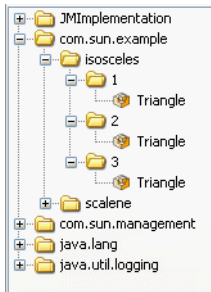


FIGURE 3-14 Example of MBean Tree Constructed Using `keyPropertyList`

In [Figure 3-14](#), the `side` key comes first, followed by the `name` key. The `type` key comes at the end because it was not specified in the key property list, so the MBean tree algorithm applied the original key order for the remaining keys. Consequently, the `type` key is appended at the end, after the keys which were defined by the `keyPropertyList` system property.

According to the object name convention defined by the [JMX Best Practices Guidelines](#), the `type` key should always come first. So, to respect this convention you should start JConsole with the following system property.

```
% jconsole -J-Dcom.sun.tools.jconsole.mbeans.keyPropertyList=type,side,name
```

The above command will cause JConsole to render the MBean tree for the `Triangle` MBeans as shown in [Figure 3-15](#).

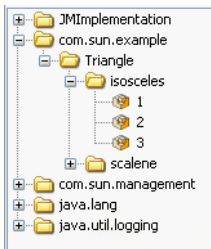


FIGURE 3-15 Example of MBean Tree Constructed Respecting JMX Best Practices

This is obviously much more comprehensible than the MBean trees shown in [Figure 3-13](#) and [Figure 3-14](#).

## MBean Attributes

Selecting the `Attributes` node displays all the attributes of an MBean. [Figure 3-16](#) shows all the attributes of the `Threading` platform `MXBean`.



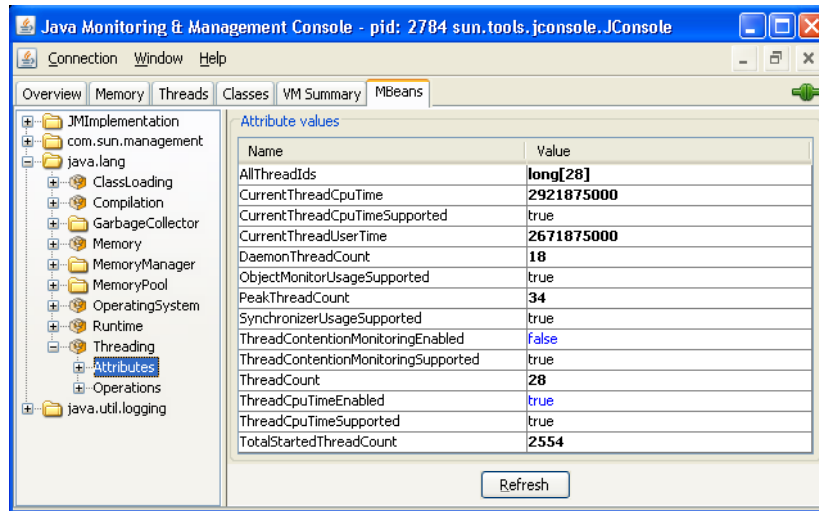


FIGURE 3-16 Viewing All MBean Attributes

Selecting an individual MBean attribute from the tree then displays the attribute's value, its `MBeanAttributeInfo`, and the associated Descriptor in the right pane, as you can see in [Figure 3-17](#).

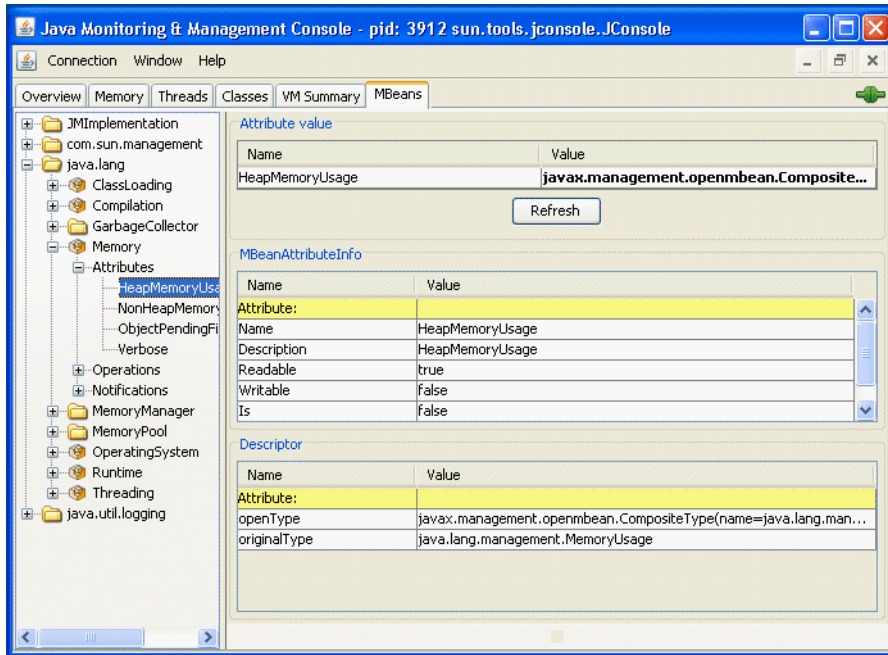


FIGURE 3-17 Viewing an Individual MBean Attribute

You can display additional information about an attribute by double-clicking on the attribute value, if it appears in bold text. For example, if you click on the value of the `HeapMemoryUsage` attribute of the `java.lang.Memory` MBean, you will see a chart that looks something like Figure 3-18.

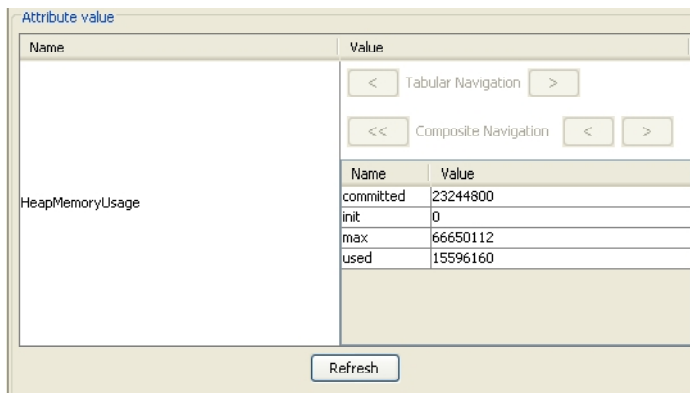


FIGURE 3-18 Displaying Attribute Values

Double-clicking on numeric attribute values will display a chart that plots changes in that numeric value. For example, double-clicking on the `CollectionTime` attribute of the Garbage Collector MBean `PSMarksweep` will plot the time spent performing garbage collection.

You can also use JConsole to set the values of writable attributes. The value of a writable attribute is displayed in blue. Here you can see the Memory MBean's `Verbose` attribute.

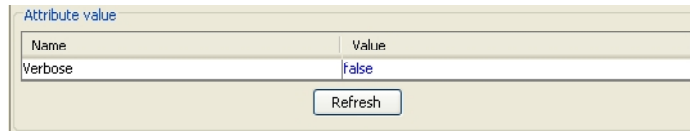


FIGURE 3-19 Setting Writable Attribute Values

You can set attributes by clicking on them and then editing them. For example, to enable or disable the verbose tracing of the garbage collector in JConsole, select the Memory MXBean in the MBeans tab and set the `Verbose` attribute to true or false. Similarly, the class loading MXBean also has the `Verbose` attribute, which can be set to enable or disable class loading verbose tracing.

## MBean Operations

Selecting the Operations node displays all the operations of an MBean. The MBean operations appear as buttons, that you can click to invoke the operation. [Figure 3-20](#) shows all the operations of the Threading platform MXBean.

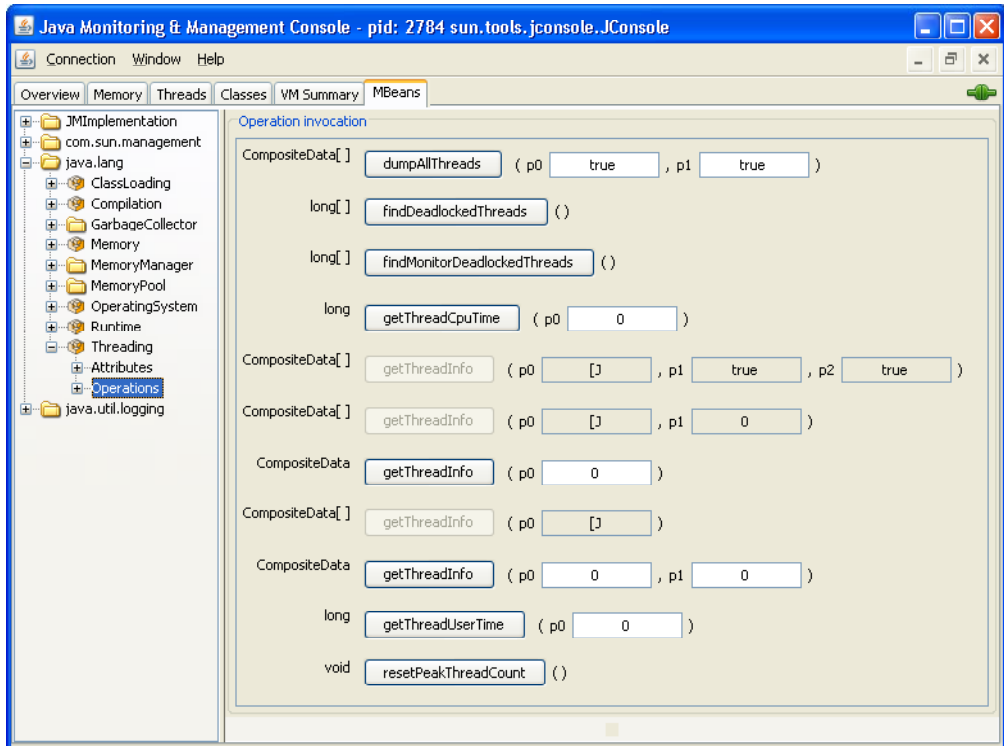


FIGURE 3–20 Viewing All MBean Operations

Selecting an individual MBean operation in the tree displays the button for invoking the MBean operation, and the operation's `MBeanOperationInfo` and its Descriptor, as shown in Figure 3–21.

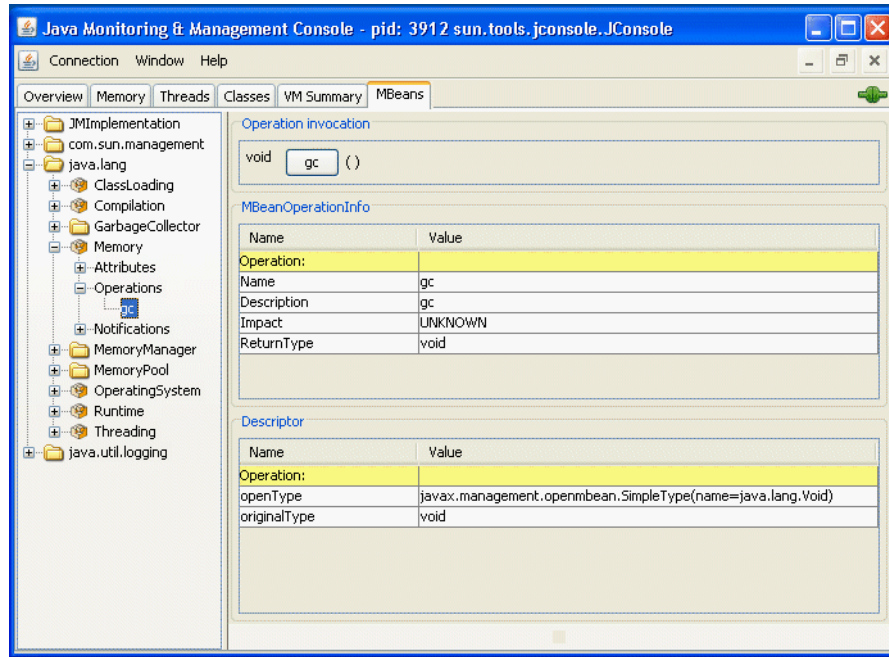


FIGURE 3-21 Viewing Individual MBean Operations

## MBean Notifications

You can subscribe to receive notifications by selecting the Notifications node in the left-hand tree, and clicking the Subscribe button that appears on the right. The number of notifications received is displayed in square brackets, and the Notifications node itself will appear in bold text when new notifications are received. The notifications of the Memory platform MXBean are shown in Figure 3-22.

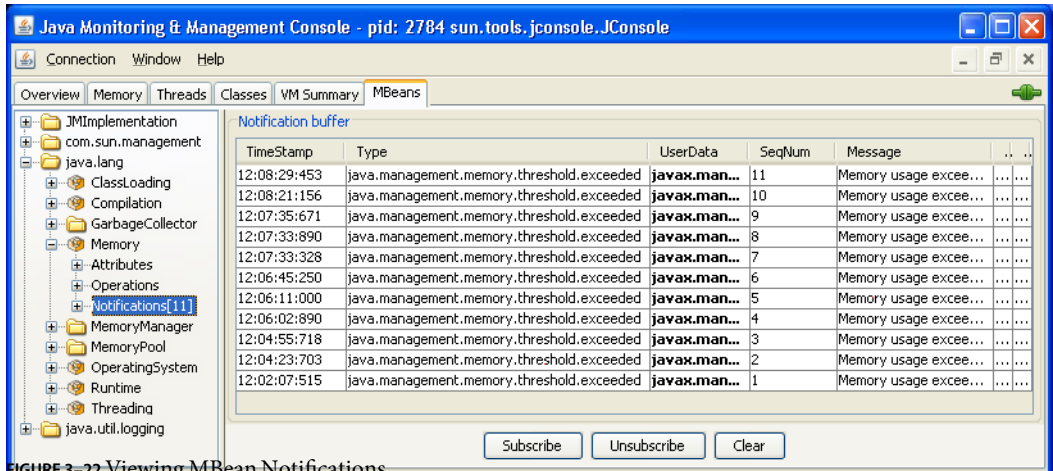


FIGURE 3–22 Viewing MBean Notifications

Selecting an individual MBean notification displays the MBeanNotificationInfo in the right pane, as shown in Figure 3–23.

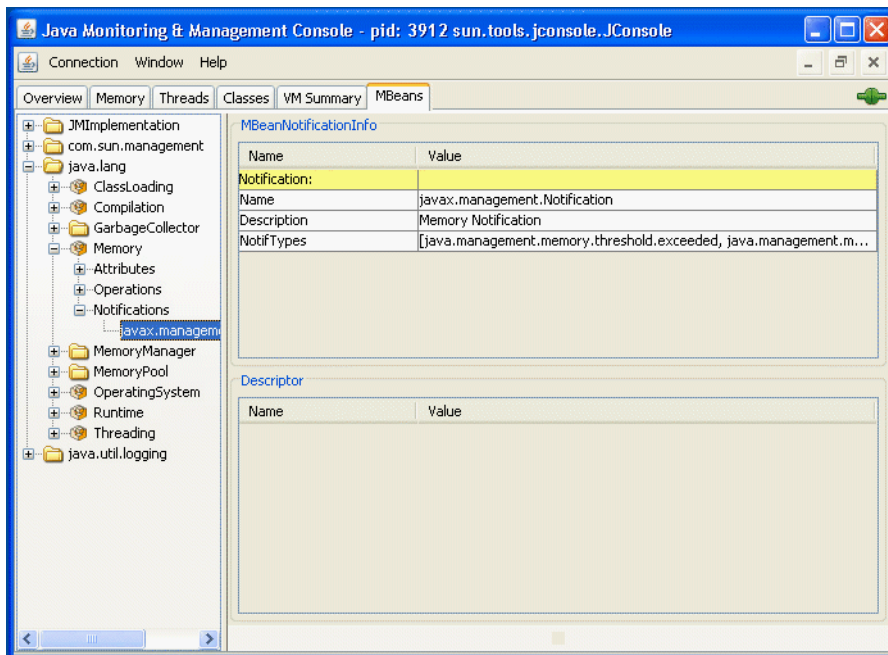


FIGURE 3–23 Viewing Individual MBean Notifications

## HotSpot Diagnostic MBean

JConsole's MBeans tab also allows you to tell the HotSpot VM to perform a heap dump, and to get or set a VM option via the HotSpotDiagnostic MBean.

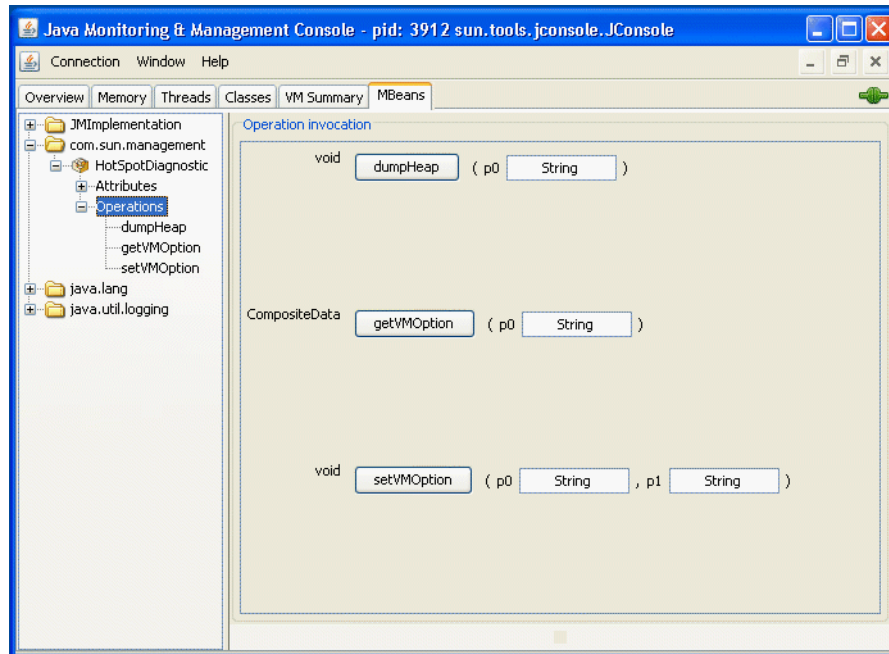


FIGURE 3-24 Viewing the HotSpot Diagnostic MBean

You can perform a heap dump manually by invoking the `com.sun.management.HotSpotDiagnostic` MBean's `dumpHeap` operation. In addition, you can specify the `HeapDumpOnOutOfMemoryError` Java VM option using the `setVMOption` operation, so that the VM performs a heap dump automatically whenever it receives an `OutOfMemoryError`.

## Creating Custom Tabs

In addition to the existing standard tabs, you can add your own custom tabs to JConsole, to perform your own monitoring activities. The JConsole plug-in API provides a mechanism by which you can, for example, add a tab to access your own application's MBeans. The JConsole plug-in API defines the `com.sun.tools.jconsole.JConsolePlugin` abstract class that you can extend to build your custom plug-in.

As stated above, your plug-in must extend `JConsolePlugin`, and implement the `JConsolePlugin` `getTabs` and `newSwingWorker` methods. The `getTabs` method returns either the list of tabs to be added to JConsole, or an empty list. The `newSwingWorker` method returns the `SwingWorker` to be responsible for the plug-in's GUI update.

Your plug-in must be provided in a Java archive (JAR) file that contains a file named `META-INF/services/com.sun.tools.jconsole.JConsolePlugin`. This `JConsolePlugin` file itself contains a list of all the fully-qualified class names of the plug-ins you want to add as new JConsole tabs. JConsole uses the service-provider loading facility to look up and load the plug-ins. You can have multiple plug-ins, with one entry per plug-in in the `JConsolePlugin`.

To load the new custom plug-ins into JConsole, start JConsole with the following command:

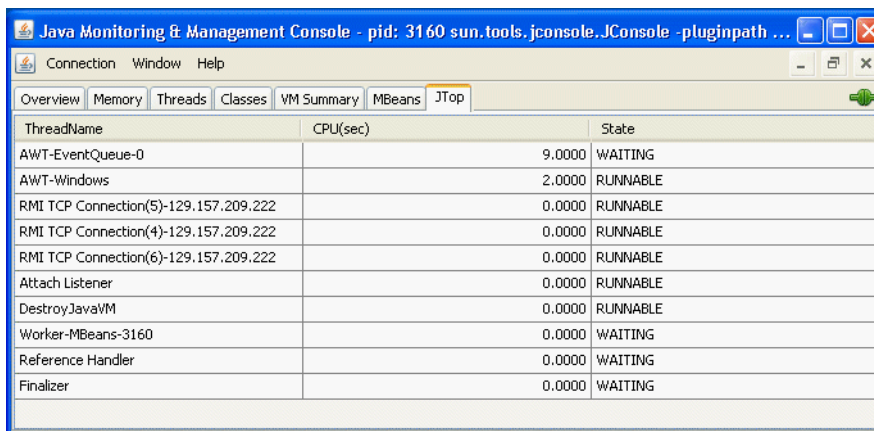
```
% jconsole -pluginpath plugin-path
```

In the above command, *plugin-path* specifies the paths to the JConsole plug-ins to be looked up. These paths can either be to directory names or to JAR files, and multiple paths can be specified, using your platform's standard separator character.

An example JConsole plug-in is provided with the Java SE 6 platform. The JTop application is a JDK demonstration that shows the CPU usage of all threads running in the application. This demo is useful for identifying threads that have high CPU consumption, and it has been updated to be used as a JConsole plug-in as well as a standalone GUI. JTop is bundled with the Java SE 6 platform, as a demo application. You can run JConsole with the JTop plug-in by running the following command:

```
% JDK_HOME/bin/jconsole -pluginpath JDK_HOME/demo/management/JTop/JTop.jar
```

If you connect to this instance of JConsole, you will see that the JTop tab has been added, showing CPU usage of the various threads running.



ThreadName	CPU(sec)	State
AWT-EventQueue-0	9.0000	WAITING
AWT-Windows	2.0000	RUNNABLE
RMI TCP Connection(5)-129.157.209.222	0.0000	RUNNABLE
RMI TCP Connection(4)-129.157.209.222	0.0000	RUNNABLE
RMI TCP Connection(6)-129.157.209.222	0.0000	RUNNABLE
Attach Listener	0.0000	RUNNABLE
DestroyJavaVM	0.0000	RUNNABLE
Worker-MBeans-3160	0.0000	WAITING
Reference Handler	0.0000	WAITING
Finalizer	0.0000	WAITING

FIGURE 3–25 Viewing a Custom Plug-in Tab



# Using the Platform MBean Server and Platform MXBeans

---

This chapter introduces the MBean server and the MXBeans that are provided as part of the Java Platform, Standard Edition (Java SE platform), which can be used for monitoring and management purposes. Java Management Extensions (JMX) technology MBeans and MBean servers were introduced briefly in [Chapter 1](#). More information about the JMX technology can be found in the [JMX Technology documentation](#) for the Java SE platform.

## Using the Platform MBean Server

An MBean server is a repository of MBeans that provides management applications access to MBeans. Applications do not access MBeans directly, but instead access them through the MBean server via their unique `ObjectName`. An MBean server implements the interface `javax.management.MBeanServer`.

The *platform MBean server* was introduced in the Java 2 Platform, Standard Edition 5.0, and is an MBean server that is built into the Java Virtual Machine (Java VM). The platform MBean server can be shared by all managed components that are running in the Java VM. You access the platform MBean server using the `java.lang.management.ManagementFactory` method `getPlatformMBeanServer`. Of course, you can also create your own MBean server using the `javax.management.MBeanServerFactory` class. However, there is generally no need for more than one MBean server, so using the platform MBean server is recommended.

## Accessing Platform MXBeans

A *platform MXBean*, is an MBean for monitoring and managing the Java VM. Each MXBean encapsulates a part of the VM functionality. A full list of the MXBeans that are provided with the platform is provided in [Table 1-1](#) in [Chapter 1](#).

A management application can access platform MXBeans in three different ways.

- Direct access, via the `ManagementFactory` class.

- Direct access, via an MXBean proxy.
- Indirect access, via the `MBeanServerConnection` class.

These three ways of accessing the platform MXBeans are described in the next three sections.

## Accessing Platform MXBeans via the ManagementFactory Class

An application can make direct calls to the methods of a platform MXBean that is running in the same Java VM as itself. To make direct calls, you can use the static methods of the `ManagementFactory` class. `ManagementFactory` has accessor methods for each of the different platform MXBeans, such as `getClassLoadingMXBean()`, `getGarbageCollectorMXBeans()`, `getRuntimeMXBean()`, and so on. In cases where there are more than one platform MXBean, the method returns a list of the platform MXBeans found.

For example, [Example 4-1](#) uses the static method of `ManagementFactory` to get the platform MXBean `RuntimeMXBean`, and then gets the vendor name from the platform MXBean.

**EXAMPLE 4-1** Accessing a Platform MXBean via the `ManagementFactory` Class

```
RuntimeMXBean mxbean = ManagementFactory.getRuntimeMXBean();
String vendor = mxbean.getVmVendor();
```

## Accessing Platform MXBeans via an MXBean Proxy

An application can also call platform MXBean methods via an MXBean proxy. To do so, you must construct an MXBean proxy instance that forwards the method calls to a given MBean server by calling the static method `ManagementFactory.newPlatformMXBeanProxy()`. An application typically constructs a proxy to obtain remote access to a platform MXBean of another Java VM.

For example, [Example 4-2](#) performs exactly the same operation as [Example 4-1](#), but this time uses an MXBean proxy.

**EXAMPLE 4-2** Accessing a Platform MXBean via an MXBean Proxy

```
MBeanServerConnection mbs;
...
// Get a MBean proxy for RuntimeMXBean interface
RuntimeMXBean proxy =
    ManagementFactory.newPlatformMXBeanProxy(mbs,
                                              ManagementFactory.RUNTIME_MXBEAN_NAME,
                                              RuntimeMXBean.class);

// Get standard attribute "VmVendor"
String vendor = proxy.getVmVendor();
```

## Accessing Platform MBeans via the MBeanServerConnection Class

An application can indirectly call platform MBean methods through an `MBeanServerConnection` that connects to the platform MBean server of another running Java VM. You use the `MBeanServerConnection` class's `getAttribute()` method to get an attribute of a platform MBean, providing the MBean's `ObjectName` and the attribute name as parameters.

For example, [Example 4-3](#) performs the same job as [Example 4-1](#) and [Example 4-2](#), but uses an indirect call through `MBeanServerConnection`.

**EXAMPLE 4-3** Accessing a Platform MBean via the `MBeanServerConnection` Class

```
MBeanServerConnection mbs;
...
try {
    ObjectName oname = new ObjectName(ManagementFactory.RUNTIME_MXBEAN_NAME);
    // Get standard attribute "VmVendor"
    String vendor = (String) mbs.getAttribute(oname, "VmVendor");
} catch (...) {
    // Catch the exceptions thrown by ObjectName constructor
    // and MBeanServer.getAttribute method
    ...
}
```

## Using Sun Microsystems' Platform Extension

Java VMs can extend the management interface by defining interfaces for platform-specific measurements and management operations. The static factory methods in the `ManagementFactory` class will return the MBeans with the platform extension.

The `com.sun.management` package contains Sun Microsystems' platform extensions. The following sections provide examples of how to access a platform-specific attribute from Sun Microsystems' implementation of the `OperatingSystemMBean`.

### Accessing MBean Attributes Directly

[Example 4-4](#) illustrates direct access to one of Sun Microsystems' MBean interfaces.

**EXAMPLE 4-4** Accessing an MBean Attribute Directly

```
com.sun.management.OperatingSystemMBean mxbean =
    (com.sun.management.OperatingSystemMBean) ManagementFactory.getOperatingSystemMBean();
```

**EXAMPLE 4-4** Accessing an MBean Attribute Directly *(Continued)*

```
// Get the number of processors
int numProcessors = mbean.getAvailableProcessors();

// Get the Sun-specific attribute Process CPU time
long cpuTime = mbean.getProcessCpuTime();
```

## Accessing MBean Attributes via MBeanServerConnection

[Example 4-5](#) illustrates access to one of Sun Microsystems' MBean interfaces via the `MBeanServerConnection` class.

**EXAMPLE 4-5** Accessing an MBean Attribute via `MBeanServerConnection`

```
MBeanServerConnection mbs;

// Connect to a running Java VM (or itself) and get MBeanServerConnection
// that has the MBeans registered in it
...

try {
    // Assuming the OperatingSystem MBean has been registered in mbs
    ObjectName oname = new ObjectName(ManagementFactory.OPERATING_SYSTEM_MXBEAN_NAME);

    // Get standard attribute "Name"
    String vendor = (String) mbs.getAttribute(oname, "Name");

    // Check if this MBean contains Sun Microsystems' extension
    if (mbs.isInstanceOf(oname, "com.sun.management.OperatingSystemMBean")) {
        // Get platform-specific attribute "ProcessCpuTime"
        long cpuTime = (Long) mbs.getAttribute(oname, "ProcessCpuTime");
    }
} catch (...) {
    // Catch the exceptions thrown by ObjectName constructor
    // and MBeanServer methods
    ...
}
```

## Monitoring Thread Contention and CPU Time

The `ThreadMXBean` platform `MXBean` provides support for monitoring thread contention and thread Central Processing Unit (CPU) time.

The Sun HotSpot VM supports thread contention monitoring. You use the `ThreadMXBean.isThreadContentionMonitoringSupported()` method to determine if a Java VM supports thread contention monitoring. Thread contention monitoring is disabled by default. Use the `setThreadContentionMonitoringEnabled()` method to enable it.

The Sun HotSpot VM supports the measurement of thread CPU time on most platforms. The CPU time provided by this interface has nanosecond precision but not necessarily nanosecond accuracy.

You use the `isThreadCpuTimeSupported()` method to determine if a Java VM supports the measurement of the CPU time for any thread. You use `isCurrentThreadCpuTimeSupported()` to determine if a Java VM supports the measurement of the CPU time for the current thread. A Java VM that supports CPU time measurement for any thread will also support that for the current thread.

A Java VM can disable thread CPU time measurement. You use the `isThreadCpuTimeEnabled()` method to determine if thread CPU time measurement is enabled. You use the `setThreadCpuTimeEnabled()` method to enable or disable the measurement of thread CPU time.

## Managing the Operating System

The `OperatingSystem` platform `MXBean` allows you to access certain operating system resource information, such as the following.

- Process CPU time.
- Amount of total and free physical memory.
- Amount of committed virtual memory (that is, the amount of virtual memory guaranteed to be available to the running process).
- Amount of total and free swap space.
- Number of open file descriptors (only for UNIX platforms).

When the `OperatingSystem` `MXBean` in the `MBeans` tab is selected in `JConsole`, you see all the attributes and operations including the platform extension. You can monitor the changes of a numerical attribute over time by double-clicking the value field of the attribute.

## Logging Management

The Java SE platform provides a special MXMLBean for logging purposes, the `LoggingMXMLBean` interface.

The `LoggingMXMLBean` enables you to perform the following tasks.

- Get the name of the log level associated with the specified logger.
- Get the list of currently registered loggers.
- Get the name of the parent for the specified logger.
- Set the specified logger to the specified new level.

The unique `ObjectName` of the `LoggingMXMLBean` is `java.util.logging:type=Logging`. This object name is stored in `LogManager.LOGGING_MXMLBEAN_NAME`.

There is a single global instance of the `LoggingMXMLBean`, which you can get by calling `LogManager.getLoggingMXMLBean()`.

The `Logging MXMLBean` defines a `LoggerNames` attribute describing the list of logger names. To find the list of loggers in your application, you can select the `Logging MXMLBean` under the `java.util.logging` domain in the MBeans tab, and double-click on the value field of the `LoggerNames` attribute. The `Logging MXMLBean` also supports two operations.

- `getLoggerLevel`: Returns the log level of a given logger.
- `setLoggerLevel`: Sets the log level of a given logger to a new level.

These operations take a logger name as the first parameter. To change the level of a logger, enter the logger name in the first parameter and the name of the level it should be set to in the second parameter of the `setLoggerLevel` operation.

## Detecting Low Memory

Memory use is an important attribute of the memory system. It can be indicative of the following problems.

- Excessive memory consumption by an application.
- An excessive workload imposed on the automatic memory management system.
- Potential memory leakages.

There are two kinds of memory thresholds you can use to detect low memory conditions: a *usage threshold* and a *collection usage threshold*. You can detect low memory conditions using either of these thresholds with *polling* or *threshold notification*. All these concepts are described in the next sections.

## Memory Thresholds

A memory pool can have two kinds of memory thresholds: a usage threshold and a collection usage threshold. Either one of these thresholds may not be supported by a particular memory pool. The values for the usage threshold and collection usage threshold can both be set using the MBeans tab in JConsole.

### Usage Threshold

The usage threshold is a manageable attribute of some memory pools. It enables you to monitor memory use with a low overhead. Setting the threshold to a positive value enables a memory pool to perform usage threshold checking. Setting the usage threshold to zero disables usage threshold checking. The default value is supplied by the Java VM.

A Java VM performs usage threshold checking on a memory pool at the most appropriate time, typically during garbage collection. Each memory pool increments a usage threshold count whenever the usage crosses the threshold.

You use the `isUsageThresholdSupported()` method to determine whether a memory pool supports a usage threshold, since a usage threshold is not appropriate for some memory pools. For example, in a generational garbage collector (such as the one in the HotSpot VM; see [“Garbage Collection” on page 47 in Chapter 3](#)), most of the objects are allocated in the young generation, from the Eden memory pool. The Eden pool is designed to be filled up. Garbage collecting the Eden memory pool will free most of its memory space since it is expected to contain mostly short-lived objects that are unreachable at garbage collection time. So, it is not appropriate for the Eden memory pool to support a usage threshold.

### Collection Usage Threshold

The collection usage threshold is a manageable attribute of some garbage-collected memory pools. After a Java VM has performed garbage collection on a memory pool, some memory in the pool will still be in use. The collection usage threshold allows you to set a value for this memory. You use the `isCollectionUsageThresholdSupported()` method of `MemoryPoolMXBean` to determine if the pool supports a collection usage threshold.

A Java VM may check the collection usage threshold on a memory pool when it performs garbage collection. Set the collection usage threshold to a positive value to enable checking. Set the collection usage threshold to zero (the default) to disable checking.

The usage threshold and collection usage threshold can be set in the MBeans tab of JConsole.

### Memory MXBean

The various memory thresholds can be managed via the platform `MemoryMXBean`. The `MemoryMXBean` defines the following four attributes.

- `HeapMemoryUsage`: A read-only attribute describing the current heap memory usage.
- `NonHeapMemoryUsage`: A read-only attribute describing non-heap memory usage.

- `ObjectPendingFinalizationCount`: A read-only attribute describing the number of objects pending for finalization.
- `Verbose`: A boolean attribute describing the Garbage Collection (GC) verbose tracing setting. This can be set dynamically. The GC verbose traces will be displayed at the location specified when you start the Java VM. The default location for GC verbose output of the Hotspot VM is `stdout`.

The Memory MXBean supports one operation, `gc`, for explicit garbage collection requests.

Details of the Memory MXBean interface are defined in the `java.lang.management.MemoryMXBean` specification.

## Memory Pool MXBean

The `MemoryPoolMXBean` platform MXBean defines a set of operations to manage memory thresholds.

- `getUsageThreshold()`
- `setUsageThreshold(long threshold)`
- `isUsageThresholdExceeded()`
- `isUsageThresholdSupported()`
- `getCollectionUsageThreshold()`
- `setCollectionUsageThreshold(long threshold)`
- `isCollectionUsageThresholdSupported()`
- `isCollectionUsageThresholdExceeded()`

Each memory pool may have two kinds of memory thresholds for low memory detection support: a usage threshold and a collection usage threshold. Either one of these thresholds might not be supported by a particular memory pool. For more information, see the API reference documentation for the `MemoryPoolMXBean` class.

## Polling

An application can continuously monitor its memory usage by calling either the `getUsage()` method for all memory pools or the `isUsageThresholdExceeded()` method for memory pools that support a usage threshold.

[Example 4-6](#) has a thread dedicated to task distribution and processing. At every interval, it determines whether it should receive and process new tasks based on its memory usage. If the memory usage exceeds its usage threshold, it redistributes outstanding tasks to other VMs and stops receiving new tasks until the memory usage returns below the threshold.

### EXAMPLE 4-6 Using Polling

```
pool.setUsageThreshold(myThreshold);  
...  
boolean lowMemory = false;
```



**EXAMPLE 4-6** Using Polling (Continued)

```

while (true) {
    if (pool.isUsageThresholdExceeded()) {
        lowMemory = true;
        redistributeTasks(); // redistribute tasks to other VMs
        stopReceivingTasks(); // stop receiving new tasks
    } else {
        if (lowMemory) { // resume receiving tasks
            lowMemory = false;
            resumeReceivingTasks();
        }
        // processing outstanding task
        ...
    }
    // sleep for sometime
    try {
        Thread.sleep(sometime);
    } catch (InterruptedException e) {
        ...
    }
}

```

[Example 4-6](#) does not differentiate the case in which the memory usage has temporarily dropped below the usage threshold from the case in which the memory usage remains above the threshold between two iterations. You can use the usage threshold count returned by `getUsageThresholdCount()` to determine if the memory usage has returned below the threshold between two polls.

To test the collection usage threshold instead, you use the `isCollectionUsageThresholdSupported()`, `isCollectionThresholdExceeded()` and `getCollectionUsageThreshold()` methods in the same way as above.

## Threshold Notifications

When the `MemoryMXBean` detects that a memory pool has reached or exceeded its usage threshold, it emits a *usage threshold exceeded* notification. The `MemoryMXBean` will not issue another usage threshold exceeded notification until the usage has fallen below the threshold and then exceeded it again. Similarly, when the memory usage after garbage collection exceeds the collection usage threshold, the `MemoryMXBean` emits a collection usage threshold exceeded notification.

[Example 4-7](#) implements the same logic as [Example 4-6](#), but uses usage threshold notification to detect low memory conditions. Upon receiving a notification, the listener notifies another thread to perform actions such as redistributing outstanding tasks, refusing to accept new tasks, or allowing new tasks to be accepted again.

In general, you should design the `handleNotification` method to do a minimal amount of work, to avoid causing delay in delivering subsequent notifications. You should perform time-consuming actions in a separate thread. Since multiple threads can concurrently invoke the notification listener, the listener should synchronize the tasks it performs properly.

**EXAMPLE 4-7** Using Threshold Notifications

```
class MyListener implements javax.management.NotificationListener {
    public void handleNotification(Notification notification, Object handback) {
        String notifType = notification.getType();
        if (notifType.equals(MemoryNotificationInfo.MEMORY_THRESHOLD_EXCEEDED)) {
            // potential low memory, redistribute tasks to other VMs & stop receiving new tasks.
            lowMemory = true;
            notifyAnotherThread(lowMemory);
        }
    }
}

// Register MyListener with MemoryMXBean
MemoryMXBean mbean = ManagementFactory.getMemoryMXBean();
NotificationEmitter emitter = (NotificationEmitter) mbean;
MyListener listener = new MyListener();
emitter.addNotificationListener(listener, null, null);
```

Assuming this memory pool supports a usage threshold, you can set the threshold to some value (representing a number of bytes), above which the application will not accept new tasks.

```
pool.setUsageThreshold(myThreshold);
```

After this point, usage threshold detection is enabled and `MyListener` will handle notification.

# SNMP Monitoring and Management

---

The Simple Network Management Protocol (SNMP) is an industry standard for network management. Objects managed by SNMP are arranged in management information bases (MIBs). The SNMP agent publishes the standard MIB for the Java virtual machine (Java VM) instrumentation. The standard MIB for monitoring and management of the Java VM is available for download at

<http://java.sun.com/javase/6/docs/jre/api/management/JVM-MANAGEMENT-MIB.mib>.

## Enabling the SNMP Agent

To monitor a Java VM with SNMP you must first enable an SNMP agent when you start the Java VM. You can enable the SNMP agent for either a single-user environment or a multiple-user environment. Then, you can monitor the Java VM with an SNMP-compliant tool.

For general information on setting system properties when you start the Java VM, see “[Setting System Properties](#)” on page 17 in [Chapter 2](#). How to enable the SNMP agent in single and multiple-user environments is described below. The process is the same for both environments, but the actions performed are slightly different.

## Access Control List File

An Access Control List (ACL) template file is provided with the Java Platform, Standard Edition (Java SE platform) in `JRE_HOME/lib/management/snmp.ac1.template`, where `JRE_HOME` is the directory in which the Java Runtime Environment (JRE) implementation is installed. You will copy this file to either `JRE_HOME/lib/management/snmp.ac1` or to your home directory, depending on whether you are operating in a single or a multiple-user environment. Ensure that only you have read permissions, since the file contains non-encrypted SNMP community strings. For security reasons, the system checks that only the owner has read permissions on the file and exits with an error if this is not the case. Thus, in a multiple-user environment, you should put this file in private location, such as your home directory.

[Example 5–1](#) shows some possible entries in an ACL file.

**EXAMPLE 5-1** Sample ACL Entries

```
#The communities public and private are allowed access from the local host.
acl = {
    {
        communities = public, private
        access = read-only
        managers = localhost
    }
}
# Traps are sent to localhost only
trap = {
    {
        trap-community = public
        hosts = localhost
    }
}
```

## ▼ To Enable the SNMP Agent in a Single-user Environment

- 1 **Set the following system property when you start the Java VM.**

```
com.sun.management.snmp.port=portNum
```

In the property above, *portNum* is the port number to use for monitoring. Setting this property starts an SNMP agent that listens on the specified port number for incoming SNMP requests.

- 2 **Create an ACL File.**

Copy the ACL template file from *JRE\_HOME/lib/management/snmp.acl.template* to *JRE\_HOME/lib/management/snmp.acl*.

- 3 **Set the permissions on the ACL file.**

Make sure the ACL file is readable by only the owner, and add community strings as needed.

## ▼ To Enable the SNMP Agent in a Multiple-user Environment

- 1 **Set the following system properties when you start the Java VM.**

```
com.sun.management.snmp.port=portNum
com.sun.management.snmp.acl.file=ACLFilePath
```

Where *ACLFilePath* is the path to the ACL file.

**2 Create an ACL File.**

Copy the ACL template file from `JRE_HOME/lib/management/snmp.acl.template` to a file named `snmp.acl` in your home directory.

**3 Set the permissions on the ACL file.**

Make sure the ACL file is readable by only the owner, and add community strings as needed.

## SNMP Monitoring and Management Properties

You can set SNMP monitoring and management properties in a configuration file or on the command line. Properties specified on the command line override properties in a configuration file. The default location for the configuration file is `JRE_HOME/lib/management.properties`. The Java VM reads this file if the command-line property `com.sun.management.snmp.port` is set.

You can specify a different location for the configuration file with the following command-line option.

```
com.sun.management.config.file=ConfigFilePath
```

In the property above, *ConfigFilePath* is the path to the configuration file.

You must specify all system properties when you start the Java VM. After the Java VM has started, any changes to system properties (for example, via the `setProperty` method), to the password file, to the ACL file, or to the configuration file will have no effect.

[Table 5-1](#) describes all the SNMP management properties.

**TABLE 5-1** SNMP monitoring and management Properties

Property Name	Description	Default
<code>com.sun.management.snmp.trap</code>	Remote port to which the SNMP agent sends traps.	162
<code>com.sun.management.snmp.interface</code>	Optional. The local host <code>InetAddress</code> , to force the SNMP agent to bind to the given <code>InetAddress</code> . This is for multi-home hosts if one wants to listen to a specific subnet only.	Not applicable
<code>com.sun.management.snmp.acl</code>	Enables or disables SNMP ACL checks.	<code>true</code>

**TABLE 5-1** SNMP monitoring and management Properties *(Continued)*

Property Name	Description	Default
<code>com.sun.management.snmp.accl.file</code>	Path to a valid ACL file. After the Java VM has started, modifying the ACL file has no effect.	<code>JRE_HOME/lib/management/snmp.accl</code>

## Configuration Errors

If any errors occur during start up of the SNMP agent, the Java VM will throw an exception and exit. Configuration errors include the following.

- Failure to bind to the port number.
- The password file is readable by anyone other than the owner.
- Invalid SNMP ACL file.

If your application runs a security manager, then additional permissions are required in the security permissions file.

# Additional Security Information For Microsoft Windows

---

## How to Secure a Password File on Microsoft Windows Systems

For remote monitoring and management, password and access files are used to control security. How to set the file permissions for a password file is described for Solaris and Linux platforms in [“Using Password and Access Files” on page 24 in Chapter 2.](#)

This appendix describes how to set the file permissions of the password file on a Windows system using a New Technology File System (NTFS) so that only the owner has read and write permissions on this file. If the file system is a File Allocation Table (FAT) 32 system, then security is not supported for this file system and the password file cannot be secured.

Securing a password file is done differently in the different versions of Windows XP. Solutions for both Windows XP Professional Edition and Windows XP Home Edition are provided in this appendix.

### ▼ To Secure a Password File on Windows XP Professional Edition

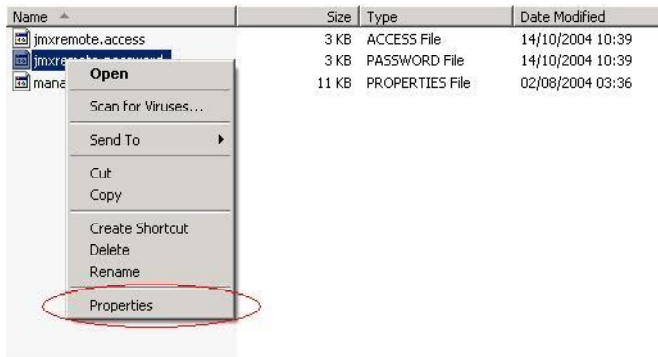
The procedure given below will not work if you are running Windows XP Home Edition, which does not allow you to change file permissions graphically. A solution is given in [“To Secure a Password File on Windows XP Home Edition” on page 86 below.](#)

---

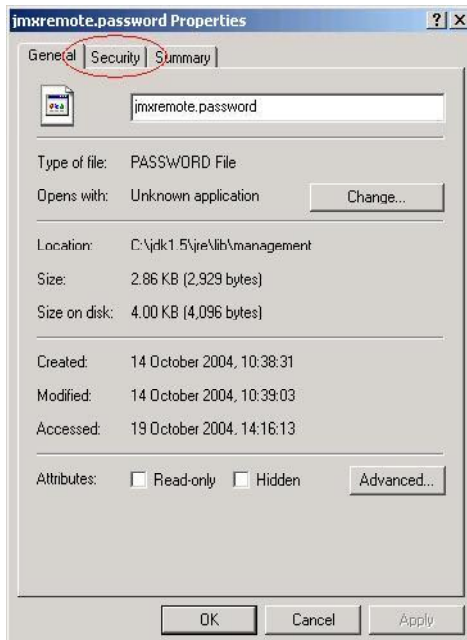
**Note** – The solution using the `cacls` command described in [“To Secure a Password File on Windows XP Home Edition” on page 86](#) can also be used on Windows XP Professional Edition, as a command-line alternative to using the graphical interfaces.

---

- 1 In Windows Explorer, navigate to the directory containing the `jmxremote.password` file.
- 2 Right-click on the `jmxremote.password` file and select the *Properties* option.



### 3 Select the Security tab

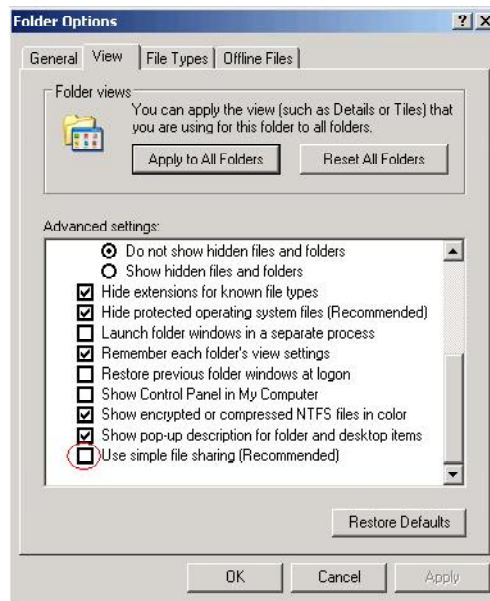


If you are using Windows XP Professional Edition and the computer is not part of a domain, then the Security tab will not be automatically visible. To reveal the Security tab, you must perform the following steps.

- a. **Open Windows Explorer, and choose *Folder Options* from the Tools menu.**

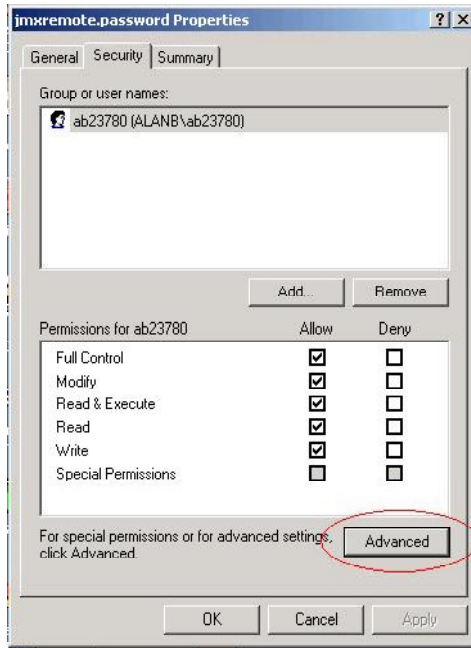


- b. Select the **View** tab and scroll to the bottom of the **Advanced Settings** and clear the *Use Simple File Sharing* check box.

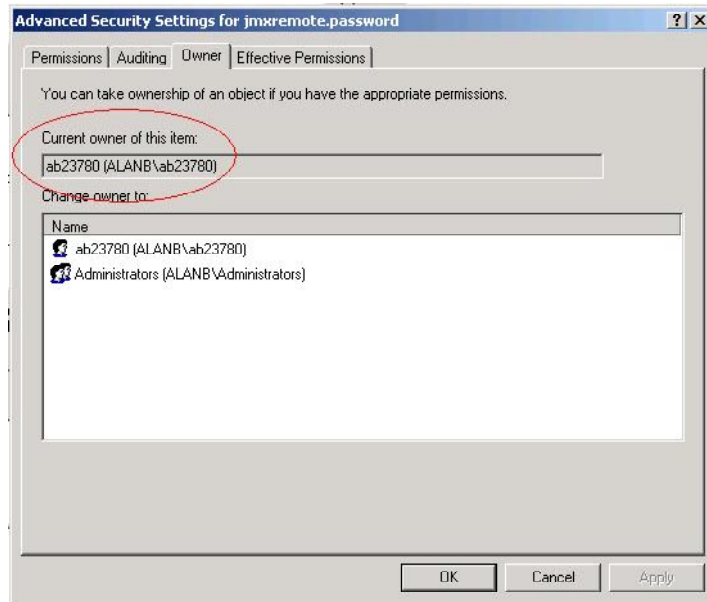


- c. Click **OK** to apply the change.
- d. Restart **Windows Explorer**.  
The **Security** tab will now be visible

- 4 Select the **Advanced** button in the **Security** tab.

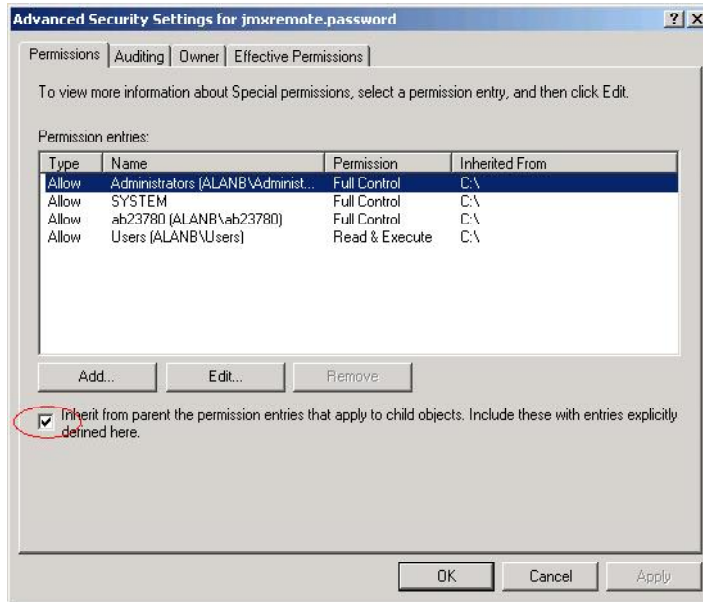


- 5 Select the Owner tab to check if the file owner matches the user under which the Java VM is running.

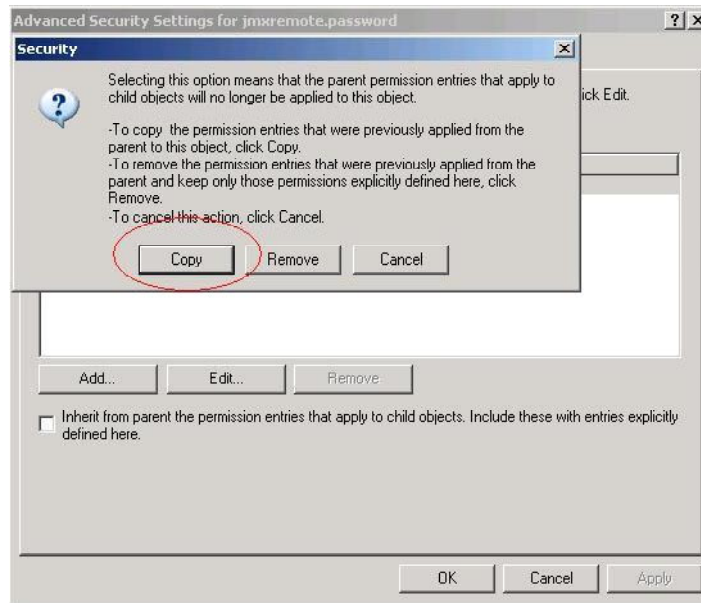


**6 Select the Permissions tab to set the permissions.**

If there are permission entries inherited from a parent directory that allow users or groups other than the owner access to the file, then clear the "Inherit from parent the permission entries that apply to child objects" checkbox.

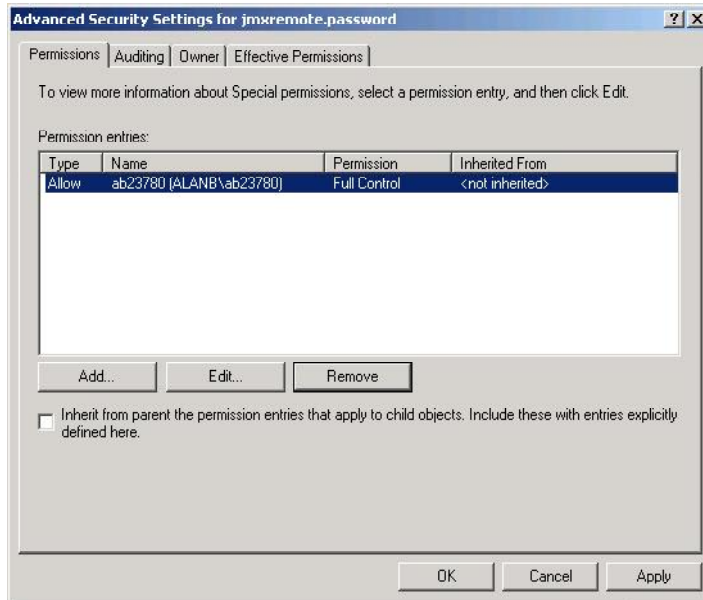


- 7 A dialog box will ask if the inherited permissions should be copied from the parent or removed. Press the Copy button.



## 8 Remove all permission entries that grant access to users or groups other than the file owner.

Do this by clicking the user or group and pressing the Remove button for all users and groups except the file owner.



Now there should be a single permission entry which grants Full Control to the owner.

**9 Press OK to apply the file security change.**

The password file is now secure and can only be accessed by the owner.

**10 Press OK in the `jmxremote.password` Properties dialog.**

## ▼ To Secure a Password File on Windows XP Home Edition

As stated above, Windows XP Home Edition does not allow you to set file permissions graphically. However, you can set permissions using the `cacls` command.

**1 Open a command prompt window.**

**2 Run the following command**

```
C:\MyPasswordFile>cacls jmxremote.password
```

This command displays the access control list (ACL) of the `jmxremote.password` file.

### 3 Set the access rights so that only your username has read access.

When no users have been configured on the machine the default username is usually Owner, or a localized translation of Owner.

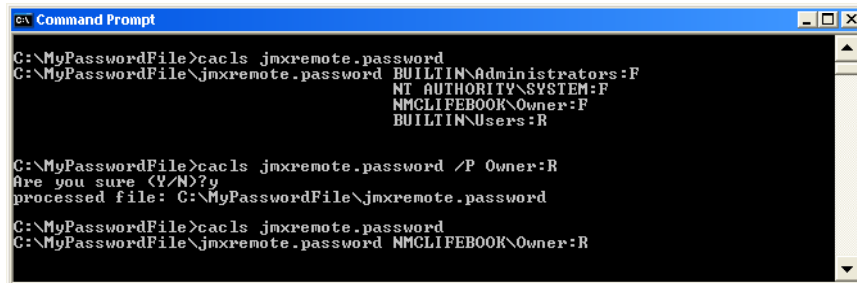
```
C:\MyPasswordFile>cac ls jmxremote.password /P Owner:R
```

This command grants access to the user Owner with read-only permission, where Owner is the owner of the `jmxremote.password` file.

### 4 Display the ACL again.

```
C:\MyPasswordFile>cac ls jmxremote.password
```

This time, you will see that only the Owner has access to the password file.



```
Command Prompt
C:\MyPasswordFile>cac ls jmxremote.password
C:\MyPasswordFile\jmxremote.password BUILTIN\Administrators:F
                                         NT AUTHORITY\SYSTEM:F
                                         NMCLIFEBOOK\Owner:F
                                         BUILTIN\Users:R

C:\MyPasswordFile>cac ls jmxremote.password /P Owner:R
Are you sure (Y/N)?y
processed file: C:\MyPasswordFile\jmxremote.password

C:\MyPasswordFile>cac ls jmxremote.password
C:\MyPasswordFile\jmxremote.password NMCLIFEBOOK\Owner:R
```

