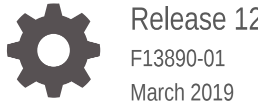
Java Platform, Standard Edition

Monitoring and Management Guide



Release 12
F13890-01
March 2019

Java Platform, Standard Edition Monitoring and Management Guide, Release 12

F13890-01

Copyright © 2006, 2019, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience vi
Documentation Accessibility vi
Related Documents vi
Conventions vi

1 Overview of Java SE Monitoring and Management

Key Monitoring and Management Features 1-1
Java Virtual Machine Instrumentation 1-1
Monitoring and Management API 1-1
Monitoring and Management Tools 1-2
Java Management Extensions Technology 1-2
What Are MBeans? 1-3
MBean Server 1-3
Creating and Registering MBeans 1-3
Instrumenting Applications 1-4
Platform MXBeans 1-4
Platform MBean Server 1-5

2 Monitoring and Management Using JMX Technology

Setting System Properties 2-1
Enabling the Ready-to-Use Management 2-1
Local Monitoring and Management 2-2
Local Monitoring and Management Using JConsole 2-2
Remote Monitoring and Management 2-3
Using Password Authentication 2-4
Disabling Password Authentication 2-6
Using SSL 2-6
Enabling RMI Registry Authentication 2-7
Enabling SSL Client Authentication 2-7
Disabling SSL 2-8

Disabling Security 2-8
Remote Monitoring with JConsole 2-8
Remote Monitoring with JConsole with SSL Enabled 2-8
Using Password and Access Files 2-9
Password Files 2-9
Access Files 2-10
Remote Monitoring with JConsole with SSL Disabled 2-10
Ready-to-Use Monitoring and Management Properties 2-11
Configuration Errors 2-13
Connecting to the JMX Agent Programmatically 2-13
Setting Up Monitoring and Management Programmatically 2-14
Mimicking Ready-to-Use Management Using the JMX Remote API 2-15
Example of Mimicking Ready-to-Use Management 2-15
Monitoring Applications Through a Firewall 2-18
Using an Agent Class to Instrument an Application 2-19
Creating an Agent Class to Instrument an Application 2-19

3 Using JConsole

Starting JConsole 3-1
Command Syntax 3-1
Setting Up Local Monitoring 3-1
Setting Up Remote Monitoring 3-2
Setting Up Secure Remote Monitoring 3-2
Connecting to a JMX Agent 3-3
Connecting JConsole to a Local Process 3-3
Connecting JConsole to a Remote Process 3-5
Connecting Using a JMX Service URL 3-6
Presenting the JConsole Tabs 3-6
Viewing Overview Information 3-7
Saving Chart Data 3-7
Monitoring Memory Consumption 3-8
Monitoring Class Loading 3-13
Viewing VM Information 3-13
Monitoring and Managing MBeans 3-15
Creating Custom Tabs 3-24

4 Using the Platform MBean Server and Platform MXBeans

Using the Platform MBean Server 4-1
Accessing Platform MXBeans 4-1

Accessing Platform MXBeans Using the ManagementFactory Class 4-1
Accessing Platform MXBeans Using an MXBean Proxy 4-2
Accessing Platform MXBeans Using the MBeanServerConnection Class 4-2
Using Oracle JDK's Platform Extension 4-3
Accessing MXBean Attributes Directly 4-3
Accessing MXBean Attributes Using MBeanServerConnection 4-3
Monitoring Thread Contention and CPU Time 4-4
Managing the Operating System 4-4
Logging Management 4-5
Detecting Low Memory 4-5
Memory Thresholds 4-5
Usage Threshold 4-6
Collection Usage Threshold 4-6
Memory MXBean 4-6
Memory Pool MXBean 4-7
Polling 4-7
Threshold Notifications 4-8

5 Java Discovery Protocol (JDP)

Preface

The Java Platform, Standard Edition 12 (Java SE 12) provide 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.

Audience

This 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.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

See JDK 12 Documentation for other JDK 12 guides.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Overview of Java SE Monitoring and Management

This topic introduces the features and utilities that provide monitoring and management services to the Java Platform, Standard Edition (Java SE platform).

Key Monitoring and Management Features

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

- Java Virtual Machine Instrumentation
- Monitoring and Management API
- Monitoring and Management Tools
- Java Management Extensions Technology

Java Virtual Machine Instrumentation

The Java Virtual Machine (Java VM) is instrumented for monitoring and management, enabling built-in (or *ready-to-use*) management capabilities that can be accessed both remotely and locally.

See Monitoring and Management Using JMX Technology.

The Java VM includes a platform MBean server and platform MBeans for use by management applications that conform to the Java Management Extensions (JMX) specification. These platforms are implementations of the monitoring and management API. The platform MXBeans and MBean servers are introduced in the Platform MXBeans and Platform MBean Server topics.

Monitoring and Management API

Java SE includes the following APIs for monitoring and management:

- `java.lang.management`: Enables monitoring and managing the Java virtual machine and the underlying operating system. The API enables applications to monitor themselves, and enables JMX-compliant tools to monitor and manage a virtual machine locally and remotely. 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
- Attach: Allows a management agent to be dynamically loaded onto a virtual machine.
- JConsole: Provides a programmatic interface to access JConsole such as adding a JConsole plug-in.

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. It provides information to help you optimize the performance.

Some of the enhancements in JConsole are as follows:

- JConsole plug-in support, which allows you to build your own plug-ins to run with JConsole. For example, you can add a custom tab for accessing the MBeans of the application.
- Dynamic attach capability allowing you to connect JConsole to any application that supports the Attach API.
- Enhanced user interface, which makes data more easily accessible.
- New Overview and VM Summary tabs for a better presentation of general information about your Java VM.
- 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 the MBeans operations and attributes.

JConsole is presented in detail in the Using JConsole topic.

Other command-line tools are also supplied with the Java SE platform.

Java Management Extensions Technology

The Java SE platform, release 12 includes the Java Management Extensions (JMX) specification, version 1.4. The JMX API allows you to instrument applications for monitoring and management. A remote method invocation (RMI) connector allows this instrumentation to be remotely accessible, for example, using JConsole.

See JMX technology documentation in the *Java Platform, Standard Edition Java Management Extensions Guide*.

The following sections provide a brief introduction to the main components of the JMX API.

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 can have attributes representing different configuration parameters, such as a `CacheSize`. Reading the `CacheSize` attribute will return the current size of the cache. Writing `CacheSize` updates the size of the cache, potentially changing the behavior of the running application. An operation such as `save` stores the current configuration persistently. The MBean can 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. An additional type of MBean, called MXBean, is 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 method 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 determines the names and types of the attributes that it exposes, by parsing an XML file.
- An *MXBean* is a type of MBean that provides a simple way to code an MBean that references only a predefined set of types. In this way, you can ensure that the MBean is usable by any client. It includes remote clients without any requirement that the client has access to model-specific classes, which represents the types of your MBeans. The platform MBeans 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 the name through the MBean server.

The Java SE platform includes a built-in platform MBean server. See Using the Platform MBean Server and Platform MXBeans.

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 method 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 the 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.

Platform MXBeans

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 MXBeans

Interface	Part of VM Managed	Object Name	Instances per VM
`ClassLoadingMXBean`	Class loading system	`java.lang:type=ClassLoading`	One
`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` (subinterface 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

Table 1-1 (Cont.) Platform MXBeans

Interface	Part of VM Managed	Object Name	Instances per VM
`RuntimeMXBean`	Runtime system	`java.lang:type=Runtime`	One
`ThreadMXBean`	Thread system	`java.lang:type=Threading`	One

The details on platform MXBeans (apart from `LoggingMXBean`) are described in the `java.lang.management` API reference. The `LoggingMXBean` interface is described in the `java.util.logging` API reference.

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, this method 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, then the platform MBean server will be created by the specified `MBeanServerBuilder` parameter.

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.

2 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 ready-to-use 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. See Table 2-1 for the full set of ready-to-use management properties. You can also set system properties in a configuration file, as described in the Ready-to-Use Monitoring and Management Properties section.

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 enter the command.

The syntax and the full set of command-line options supported by the Java HotSpot VMs are described in the Java application launcher section of *Java Platform, Standard Edition Tools Reference*.

Enabling the Ready-to-Use 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 with the JMX specification, such as JConsole. See Using JConsole.

Local Monitoring and Management

Earlier while starting the Java VM or Java application, you set the following property to allow the JMX client access to a local Java VM:

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

Setting this property registered the Java VM platform's MBeans and published the remote method invocation (RMI) connector through a private interface. This setting allows 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 current Java SE platform, it is no longer necessary to set this system property. Any application that is started on the current Java SE platform supports the Attach API, and 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 run the following commands:

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

In the preceding command, *`JDK_HOME`* is the directory in which the Java Development Kit (JDK) is installed. In the current Java SE platform, you have to run the following command to start `Notepad`.

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

After `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 supported only 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, then you start the tool by entering `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 that you want to monitor. Both JConsole and the application must by executed by the same user, because the monitoring and management 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 enter the command. See Using JConsole.

Remote Monitoring and Management

By default, the remote stubs for locally created remote objects that are sent to client contains the IP address of the local host in `dotted-quad` format. For remote stubs to be associated with a specific interface address, the `java.rmi.server.hostname` system property must be set to IP address of that interface.

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
```

Where, *portNum* is the port number to enable JMX RMI connections. Ensure that you 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 the name, `jmxrmi`. The port number to which the RMI connector will be bound using the system property:

```
com.sun.management.jmxremote.rmi.port
```

Ensure to use an unused port number.

Note:

You must set the prior system property in addition to any properties that you might set for security.

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.

Note:

For production systems, use both SSL client certificates to authenticate the client host and password authentication for user management. See Using SSL and Using LDAP Authentication.

The Java platform supports pluggable `login modules` for authentication. You can plug in any login module depending on the authentication infrastructure in your organization. Using LDAP Authentication describes how to plug in the `com.sun.security.auth.module.LdapLoginModule` module for Lightweight Directory Access Protocol (LDAP)-based authentication.

After you have enabled the JMX agent for remote use, you can monitor your application using JConsole, as described in Remote Monitoring with JConsole. How to connect to the management agent programmatically is described in Connecting to the JMX Agent Programmatically.

Using Password Authentication

This section details different password authentication methods that can be implemented based on the requirement.

Using LDAP Authentication

The `JMXAuthenticator` implementation in the JMX agent is based on Java Authentication and Authorization Service (JAAS) technology. Authentication is performed by passing the user credentials to a JAAS `javax.security.auth.spi.LoginModule` object. The `com.sun.security.auth.module.LdapLoginModule` class enables authentication using LDAP. You can replace the default `LoginModule` class with the `LdapLoginModule` class.

Create a JAAS configuration file that works in the required business organization. Here is an example of a configuration file (`ldap.config`) :

```
ExampleCompanyConfig {
    com.sun.security.auth.module.LdapLoginModule REQUIRED
        userProvider="ldap://example-ds/ou=people,dc=examplecompany,dc=com"
        userFilter="(&(uid={USERNAME})(objectClass=inetOrgPerson))"
        authzIdentity=monitorRole;
    };
```

Here is an overview of the options mentioned in the configuration file:

- The `com.sun.security.auth.module.LdapLoginModule REQUIRED` option means that authentication using `LdapLoginModule` is required for the overall authentication to be successful.
- The `userProvider` option identifies the LDAP server and the position in the directory tree where user entries are located.
- The `userFilter` option specifies the search filter to use to locate a user entry in the LDAP directory. The token `{USERNAME}` is replaced with the user name before the filter is used to search the directory.

- The `authzIdentity` option specifies the access role for authenticated users. In the example, authenticated users will have the `monitorRole` option. See Access Files.

The details of the configuration options mentioned in the code example is explained in the `com.sun.security.auth.module.LdapLoginModule` class.

Start your application with the following properties set on the command line:

- `com.sun.management.jmxremote.login.config`: This property configures the JMX agent to use the specified JAAS configuration entry.
- `java.security.auth.login.config`: This property specifies the path to the JAAS configuration file.

Here is a sample command line:

```
java -Dcom.sun.management.jmxremote.port=5000
     -Dcom.sun.management.jmxremote.login.config=ExampleCompanyConfig
     -Djava.security.auth.login.config=ldap.config
     -jar MyApplication.jar
```

Using File-Based Password Authentication

The file-based password authentication mechanism supported by the JMX agent stores the password in clear-text and is intended only for development use. For production use, it is recommended that you use SSL client certificates for authentication or plug in a secure login configuration.

Note:

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 start 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.

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.

As 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`. See Using Password and Access Files.

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 preceding property, *pwFilePath* is the path to the password file.

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
```

Note:

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. You can also disable passwords, but use SSL client authentication, as described in Enabling SSL Client Authentication.

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 to set up SSL is as follows:

1. If you do not have a key pair and certificate set up on the server, then perform the following tasks:
   - 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 the Importing Certificates in `keytool` documentation.

2. Configure SSL on the server system. Complete explanation of configuring and customizing SSL is beyond the scope of this document, but you generally need to set the system properties as described in the following list:

`javax.net.ssl.keyStore`	Keystore location
`javax.net.ssl.keyStoreType`	Default keystore type
`javax.net.ssl.keyStorePassword`	Default keystore password
`javax.net.ssl.trustStore`	Truststore location
`javax.net.ssl.trustStoreType`	Default truststore type
`javax.net.ssl.trustStorePassword`	Default truststore password

Setting system properties is detailed in the Setting System Properties section.

See:

- keytool - Key and Certificate Management Tool in the*Java Platform, Standard Edition Tools Reference*
- Customizing the Default Keystores and Truststores, Store Types, and Store Passwords in *Java Platform, Standard Edition Security Developer's 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 ready-to-use management agent when the Java VM is started. The default value of this property is `false`. However, it is recommended that you set this property to `true`. If this property is set to `true`, then to have full security, you must also enable SSL client authentication.

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 (default is set to `false`) to use client SSL authentication. It is recommended that you set this property to `true`. This configuration requires that the client system have a valid digital certificate. You must install a certificate and configure SSL on the client system, as described in Using SSL. 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.

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
```

Note:

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 that 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 remote client can 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 Enabled

To monitor a remote application with SSL enabled, you need to set up the `truststore` file on the system where JConsole is running and configure SSL properly. For

example, you can create a `keystore` file 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
```

See Customizing the Default Keystores and Truststores, Store Types, and Store Passwords in the *Java Platform, Standard Edition Security Developer's Guide*.

If you create the `keystore` file and start the `Server` applicaton, then start JConsole as follows:

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

See Using JConsole.

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

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` in 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, because it contains the passwords in clear-text. For security reasons, the system checks that the file is readable only 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 a private location such as your home directory.

Property names are roles, and the associated value is the role's password. Example 2–1 shows sample entries in the password file.

Example 2-1 An Example Password File

```
# specify actual password instead of the text password
monitorRole password
controlRole password
```

On Solaris, Linux, or macOS operating systems, you can set the file permissions for the password file by running the following command:

```
chmod 600 jmxremote.password
```

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 either be `readonly` or `readwrite`.

The access file defines roles and their access levels. By default, the access file defines the 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`: Grants access to read the 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`: Grants access to read and write the MBean's attributes, to call operations on them, and to create or remove them. This access should be granted only to trusted clients, because 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, then it has no access. If a role has multiple entries, then the last entry takes precedence. Typical predefined roles in the access file resemble what is shown in the Example 2-2.

Example 2-2 An Example Access File

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

Remote Monitoring with JConsole with SSL Disabled

To monitor a remote application with SSL disabled, start the 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.

Ready-to-Use Monitoring and Management Properties

You can set ready-to-use 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 is set:

- `com.sun.management.jmxremote`

  or
- `com.sun.management.jmxremote.port`

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

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

`ConfigFilePath` is the path to the configuration file.

Table 2-1 describes the ready-to-use monitoring and management properties.

Table 2-1 Ready-to-Use Monitoring and Management Properties

Property	Description	Values
`com.sun.management.jmxremote`	Enables the JMX remote agent and local monitoring using a JMX connector. This agent is published on a private interface that is used by JConsole and any other local JMX clients, which use the Attach API. JConsole can use this connector if it is started by the same user who started the agent. No password or access files are checked for requests coming from this connector.	`true` / `false`. Default is `true`.
`com.sun.management.jmxremote.port`	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 `com.sun.management.jmxremote` property.	Port number. No default.
`com.sun.management.jmxremote.registry.ssl`	Binds the RMI connector stub to an RMI registry that is protected by SSL.	`true` / `false`. Default is `false`.

Table 2-1 (Cont.) Ready-to-Use Monitoring and Management Properties

Property	Description	Values
`com.sun.management.jmxremote.ssl`	Enables secure monitoring using SSL. If the value is `false`, then SSL is not used.	`true` / `false`. Default is `true`.
`com.sun.management.jmxremote.ssl.enabled.protocols`	Shows a comma-delimited list of SSL/TLS protocol versions to enable. Used in conjunction with `com.sun.management.jmxremote.ssl`.	Default SSL/TLS protocol version.
`com.sun.management.jmxremote.ssl.enabled.cipher.suites`	Shows a comma-delimited list of SSL/TLS cipher suites to enable. Used in conjunction with `com.sun.management.jmxremote.ssl`.	Default SSL/TLS cipher suites.
`com.sun.management.jmxremote.ssl.need.client.auth`	Performs client authentication if this property is `true` and the property `com.sun.management.jmxremote.ssl` is also `true`. It is recommended that you set this property to `true`.	`true` / `false`. Default is `false`.
`com.sun.management.jmxremote.authenticate`	Prevents JMX from using password or access files if this property is `false`. All users are provided complete access.	`true` / `false`. Default is `true`.
`com.sun.management.jmxremote. password.file`	Specifies the location for the password file. If `com.sun.management.jmxremote.authenticate` is `false`, 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.	*`JRE_HOME`*`/lib/management/jmxremote.password`
`com.sun.management.jmxremote.access.file`	Specifies the location for the access file. If `com.sun.management.jmxremote.authenticate` 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.	*`JRE_HOME`*`/lib/management/jmxremote.access`

Table 2-1 (Cont.) Ready-to-Use Monitoring and Management Properties

Property	Description	Values
`com.sun.management.jmxremote.login.config`	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 `javax.security.auth.callback.NameCallback` and `javax.security.auth.callback.PasswordCallback`.	Default login configuration is a file-based password authentication.
`com.sun.management.jmxremote.rmi.port`	Specifies the port number to which the RMI connector will be bound.	Port number. Ensure to use an unused port number.

Configuration Errors

If any errors occur during the start up of the MBean server, the RMI registry, or the connector, then 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

After 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, as shown in the 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

You can create a JMX client that uses the Attach API to enable ready-to-use monitoring and management of any applications that are started on the Java SE 10 platform, without having to configure the applications for monitoring when you start them. The Attach API provides a way for tools to attach to and start agents in the target application. After an agent is running, JMX clients (and other tools) are able to obtain the JMX connector address for that agent using 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
String connectorAddress =
    vm.getAgentProperties().getProperty(CONNECTOR_ADDRESS);

// no connector address, so we start the JMX agent
if (connectorAddress == null) {
     vm.startLocalManagementAgent();

   // agent is started, get the connector address
   connectorAddress =
       vm.getAgentProperties().getProperty(CONNECTOR_ADDRESS);
}
// establish connection to connector server
JMXServiceURL url = new JMXServiceURL(connectorAddress);
JMXConnector 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`, which you can use to connect to the local JMX agent.

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

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.

> **Note:**
>
> Previous to JDK 11, the Attach API had issues locating JVMs running in docker containers. This is now fixed, and `jcmd` and `jps` work as expected. However, `jmc` will not list `java` processes running in separate docker containers. There is no known way to explicitly provide the PID of the `java` process to this tool.

Mimicking Ready-to-Use Management Using the JMX Remote API

The remote access to the ready-to-use management agent is protected by authentication and authorization, and by SSL encryption. The configuration is performed by setting system properties or by defining a `management.properties` file. In most cases, using the ready-to-use management agent and configuring it through the `management.properties` file is 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 multihomed systems. For such cases, the behavior of the ready-to-use management agent can be mimicked by using the JMX Remote API directly to create, configure, and deploy the management agent programmatically.

Example of Mimicking Ready-to-Use Management

This section provides an example of how to implement a JMX agent that identically mimics an ready-to-use management agent. In exactly the same way as the ready-to-use management agent, the agent created in Example 2-5 will run on port 3000. It 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. Information about

how to set up the SSL configuration is explained in Creating a Keystore to Use with JSSE section of *Java Platform, Standard Edition Security Developer's Guide*.

To enable monitoring and management on an application named `com.example.MyApp`, using the ready-to-use JMX agent with the configuration, run the `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 can be 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` is required to specify the location of the `management.properties` file.

Example 2-5 shows the code that you need to write to programmatically create a JMX agent, which will allow exactly the same monitoring and management on `com.example.MyApp` as using the prior command.

Example 2-5 Mimicking a Ready-to-Use 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.*;
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
        // 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 that the
        // ready-to-use management agent uses to register the RMIServer
        // stub.
        //
        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 monitored and managed in exactly the same way as if the Java platform's ready-to-use management agent has been used. However, there is one slight but important difference between the RMI registry used by the ready-to-use management agent and the one used by a management agent that mimics it. The RMI registry used by the ready-to-use management agent is read-only, namely a single entry can be bound to it and upon being bound, this entry cannot be unbound. This is not true with 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 that 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 nontrusted client.

RMI registries that 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 ready-to-use JMX agent, but does not replicate all the existing properties in the `management.properties` file. However, you can add other properties by modifying `com.example.MyApp` appropriately.

Monitoring Applications Through a Firewall

The code in Example 2-5 can be used to monitor applications through a firewall, which might not be possible if you use the ready-to-use 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 `JMXServiceURL` as follows:

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

`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 that 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.

Creating an Agent Class to Instrument an Application

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

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 is same as the `com.example.MyApp` class as 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 that 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 provide the keystore and password information because the RMI connector server is protected by SSL.

3

Using JConsole

The JConsole graphical user interface is a monitoring tool that complies with 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.

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 are taken from an instance of the interface running on Windows XP.

Starting JConsole

The `jconsole` executable file 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, then you can start JConsole by simply entering `jconsole` in a command (shell) prompt. Otherwise, you have to enter 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.

See `jconsole` in the *Java Platform, Standard Edition Tools Reference* for the complete syntax.

Setting Up Local Monitoring

Start JConsole using the following command:

```
% jconsole
```

When JConsole starts, select the required Java applications running locally that JConsole can connect to.

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

```
% jconsole processID
```

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

- On Solaris, Linux, or macOS 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 `jps` in *Java Platform, Standard Edition Tools Reference*.

For example, if the process ID of the `Notepad` application is `2956`, then start JConsole with the following command:

```
% jconsole 2956
```

Both JConsole and the application must by 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).

See Local Monitoring and Management.

Setting Up Remote Monitoring

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

```
% jconsole hostName:portNum
```

The `hostName` is the name of the system running the application and `portNum` is the port number you specified when you enabled the JMX agent while starting the Java VM. See Remote Monitoring and Management.

If you do not specify a host name/port number combination, then JConsole will display a connection dialog box (Connecting to a JMX Agent) 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). See Remote Monitoring with JConsole with SSL Enabled for the command to start JConsole with a secure connection.

Connecting to a JMX Agent

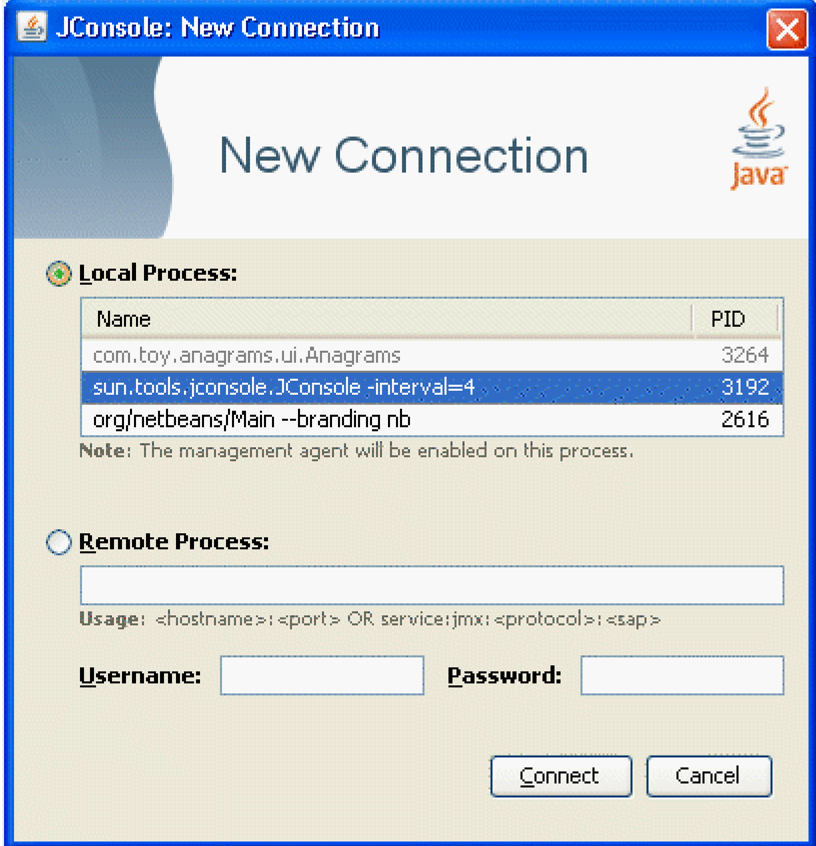
If you start JConsole with arguments specifying a JMX agent to connect to, then it will automatically start monitoring the specified Java VM. You can connect to a different host at any time by selecting **Connection** and **New Connection**, and entering the necessary information.

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

Connecting JConsole to a Local Process

If you start JConsole without providing a specific JMX agent to connect to, then you will see the following dialog box:

Figure 3-1 Creating a Connection to a Local Process



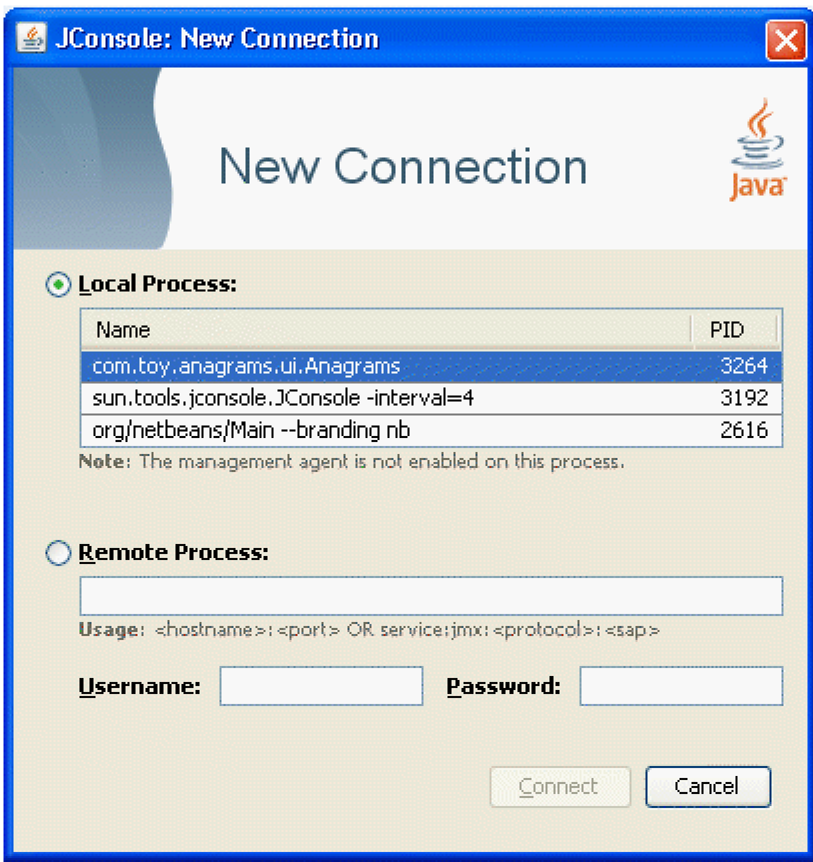
The Local Process option lists any Java VMs running on the local system that were started with the same user ID as JConsole, along with their process ID and their class or argument information. To connect JConsole to your application, select the application that you want to monitor, then click **Connect**. 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 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 platform without any management properties, but are attached to by JConsole, which enables the management agent at runtime.

- Applications that are attachable, with the management agent disabled: These include an *attachable* application that supports loading of the management agent at runtime. Attachable applications include applications that are started on the Java SE platform, which support the Attach API. Applications that 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. JConsole does not need to connect to the management agent before the application is started. If you select this application, then a note is displayed on screen that the management agent will be enabled when the connection is made. In the example, connection dialog box that is shown in Figure 3-1, the NetBeans IDE and JConsole are started within a Java SE 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 Java SE 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 box shown in Figure 3-1, the `Anagrams` application was started with a Java SE 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

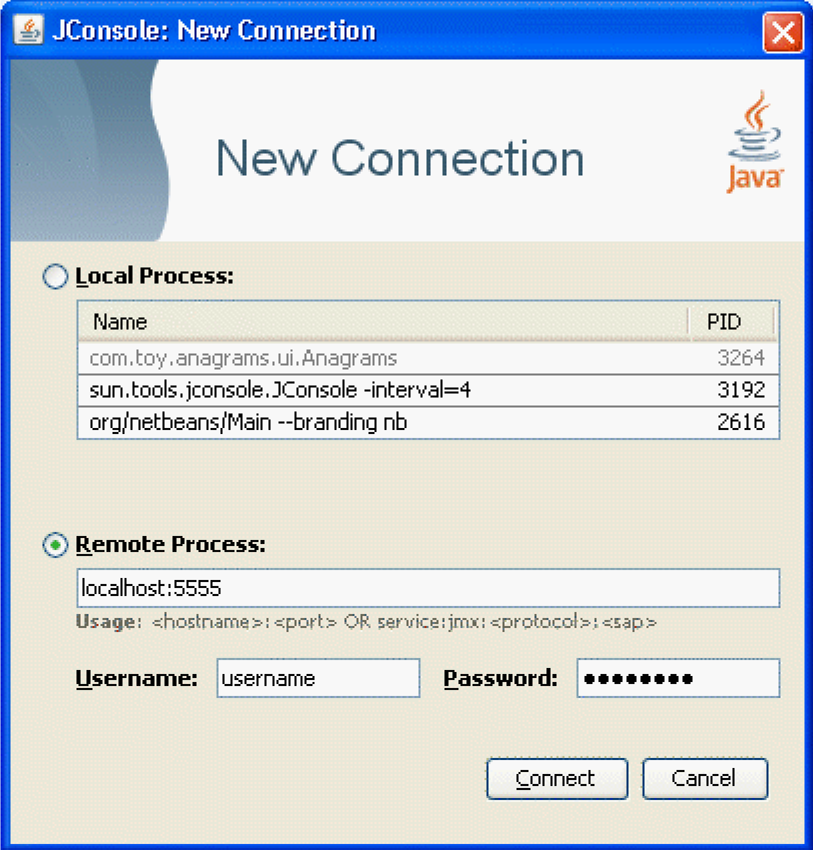


In the example connection dialog box shown in Figure 3-2, you can see that the `Anagrams` application is selected, but **Connect** remains grayed-out, and a note has appeared informing you that the management agent is not enabled for this process. JConsole cannot connect to `Anagrams` because it was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

Connecting JConsole to a Remote Process

When the connection dialog box 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: The 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).

To set the port number of the JMX agent, see Enabling the Ready-to-Use Management.

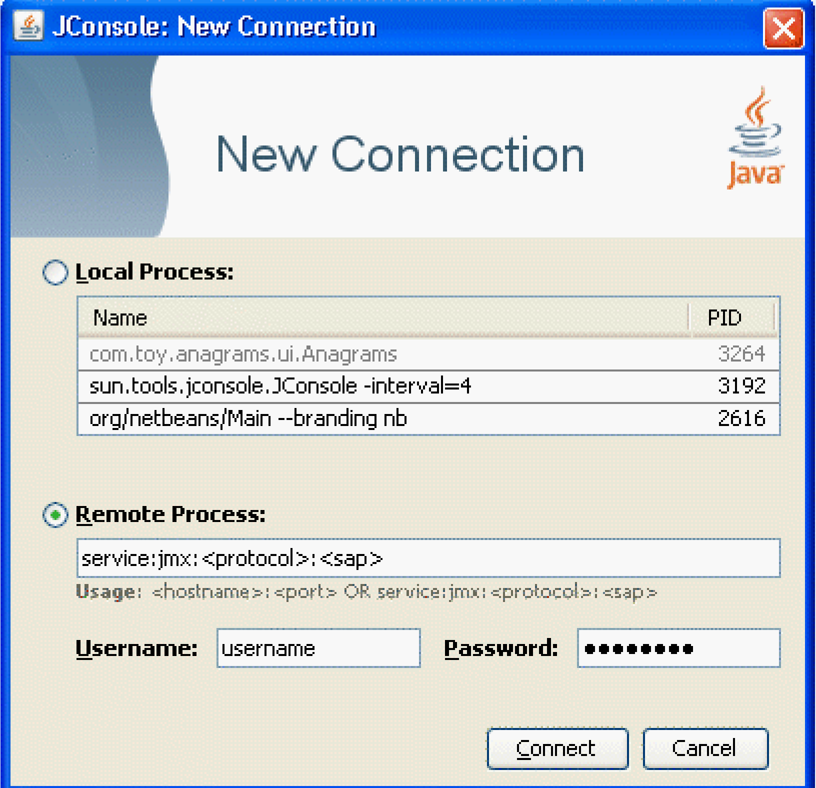
See Using Password and Access Files.

To monitor the Java VM that is running JConsole, click **Connect** and enter host as `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 that is not included in the Java platform, then you must 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
```

`connector-path` is the directory or the Java archive (JAR) file containing the connector classes that are not included in the JDK, to be used by JConsole.

Presenting the JConsole Tabs

After you have connected JConsole to an application, JConsole displays the following six tabs:

- Overview: Displays overview information about the Java VM and monitored values

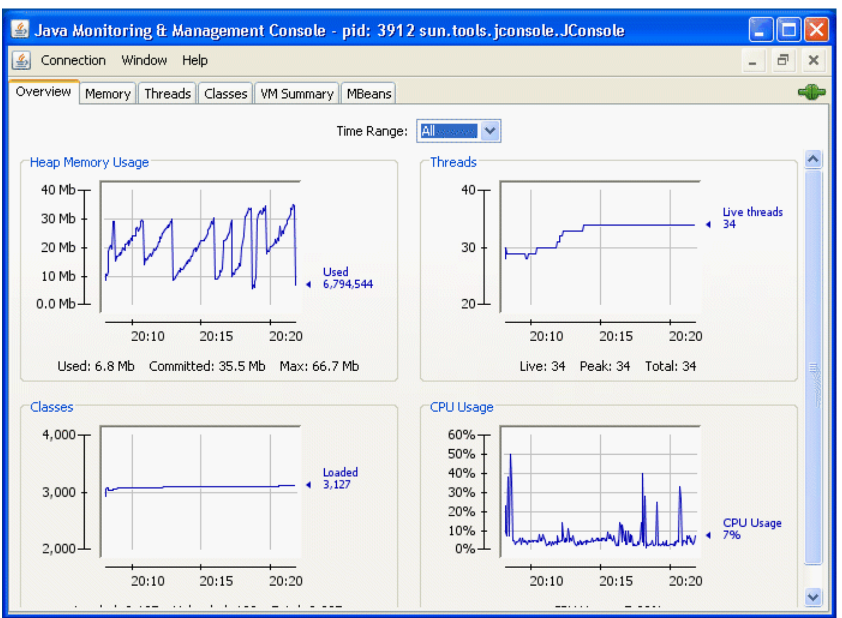
- Memory: Displays information about memory use
- Threads: Displays information about thread use
- Classes: Displays information about class loading
- VM: Displays information about the Java VM
- MBeans: Displays information about MBeans

Use the green connection status icon in the upper right-hand corner of JConsole 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 available only by switching between multiple tabs.

Saving Chart Data

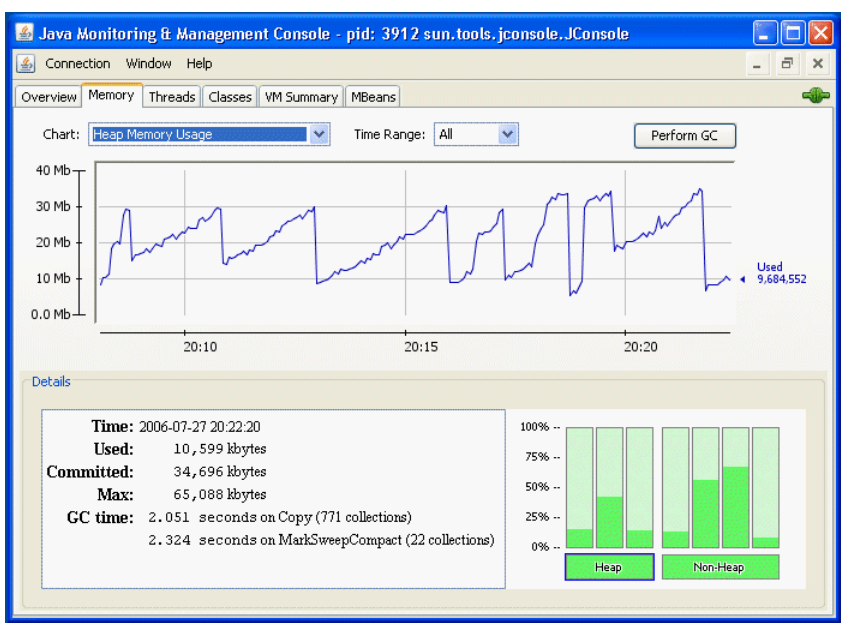
JConsole allows you to save the data presented in the charts in a comma-separated values (CSV) file. To save data from a chart, right-click on any chart, select **Save data as...**, and then specify the file in which the data will be saved. You can save the data from any of the charts displayed in any 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



Click **Perform GC** in the Memory tab to perform garbage collection whenever you want. The chart shows the memory use of the Java VM over time, for heap and nonheap memory, as well as for specific memory pools. The memory pools available depend on the version of the Java VM 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 (nonheap): 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 (nonheap): 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 selecting the required options in the **Chart** drop-down menu. Also, clicking either the Heap or Nonheap 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 the required options in the **Time Range** drop-down menu.

See Garbage Collection.

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 startup. 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 calls. 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 nonheap 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 Nonheap Memory

The Java VM manages two kinds of memory: heap and nonheap 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.
- Nonheap 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 nonheap 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.

- 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 nonheap memory.

- 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 *live* and nonreferenced (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:

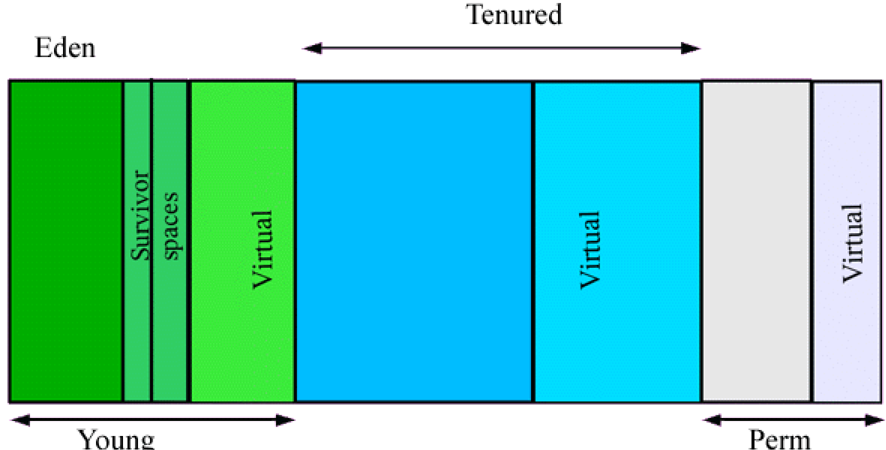
- They create many objects that have short lives, for example, iterators and local variables.
- They create some objects that have very long lives, for example, high-level persistent objects.

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

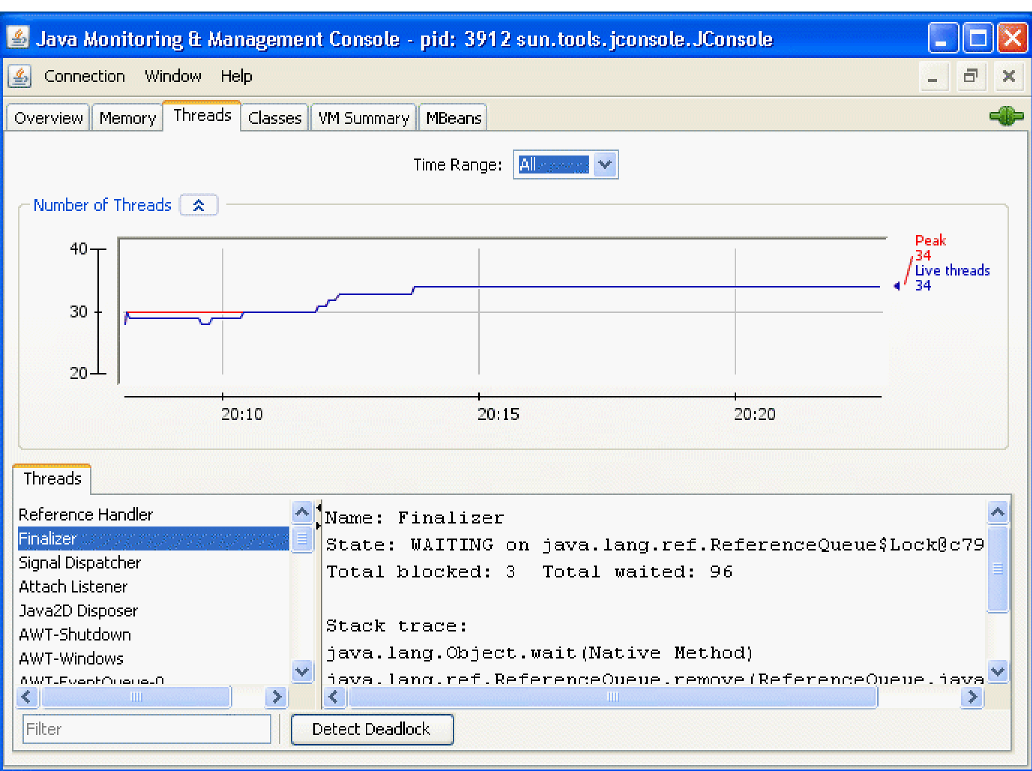


If the garbage collector has become a bottleneck, then you can improve performance by customizing the generation sizes. Using JConsole, you can investigate the sensitivity of your performance metric by experimenting with the garbage collector parameters. See Performance Considerations in *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*.

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, then the Threads list will show only those threads whose name contains the string that you entered. Click 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 MXBean 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 through the MBeans tab by selecting **Threading MXBean** in the MBeans tree. This MXBean lists all the attributes and operations for accessing threading information in the Java VM being monitored. See Monitoring and Managing MBeans.

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 **Detect Deadlock**. 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

```
Threads | Deadlock 1 | Deadlock 2

Deadlock-Thread-3
Deadlock-Thread-1
Deadlock-Thread-2

Name: Deadlock-Thread-3
State: WAITING on java.util.concurrent.locks.ReentrantLock$NonfairSyn
Total blocked: 0  Total waited: 2

Stack trace:
sun.misc.Unsafe.park(Native Method)
java.util.concurrent.locks.LockSupport.park(LockSupport.java:145)
java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAndCheckInt
java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireQueued(A
java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(Abstrac
java.util.concurrent.locks.ReentrantLock$NonfairSync.lock(ReentrantLo
java.util.concurrent.locks.ReentrantLock.lock(ReentrantLock.java:263)
SynchronizerDeadlock$DeadlockingThread.g(SynchronizerDeadlock.java:95
SynchronizerDeadlock$DeadlockingThread.f(SynchronizerDeadlock.java:88
SynchronizerDeadlock$DeadlockingThread.run(SynchronizerDeadlock.java:
```

Detect Deadlock 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 from version 6.0. If JConsole connects to a Java SE 5.0 VM, then the Detect Deadlock mechanism will find only

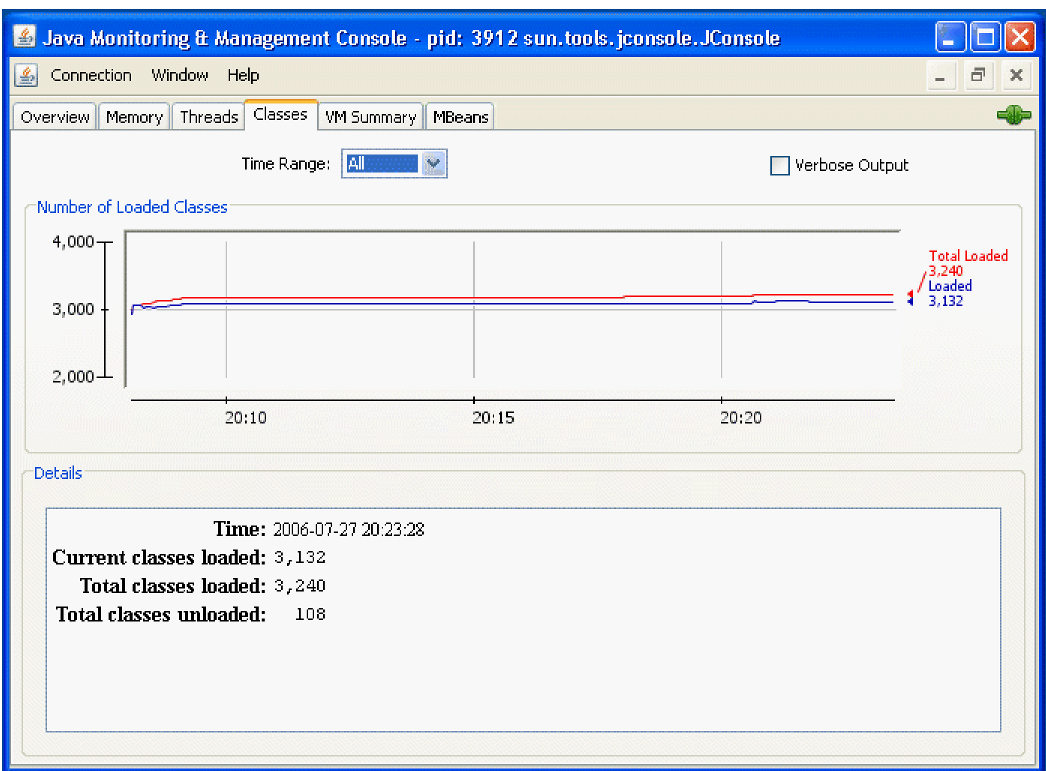
deadlocks related to object monitors. JConsole will not show any deadlocks related to ownable synchronizers.

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.

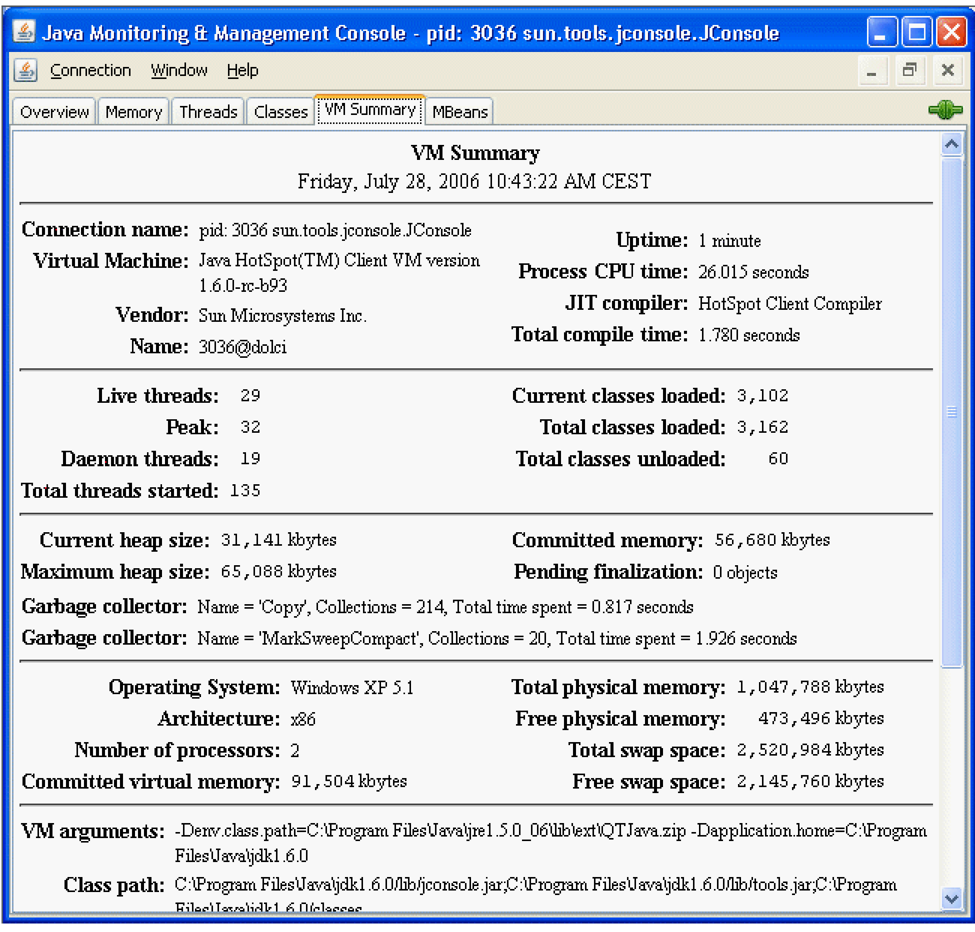
- The red line is the total number of classes loaded (including those subsequently unloaded).
- The blue line is the current number of classes loaded.

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 selecting the check box 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 nondaemon 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, nondaemon, and terminated threads.
- **Classes**
  - Current classes loaded: Number of classes currently loaded into memory.

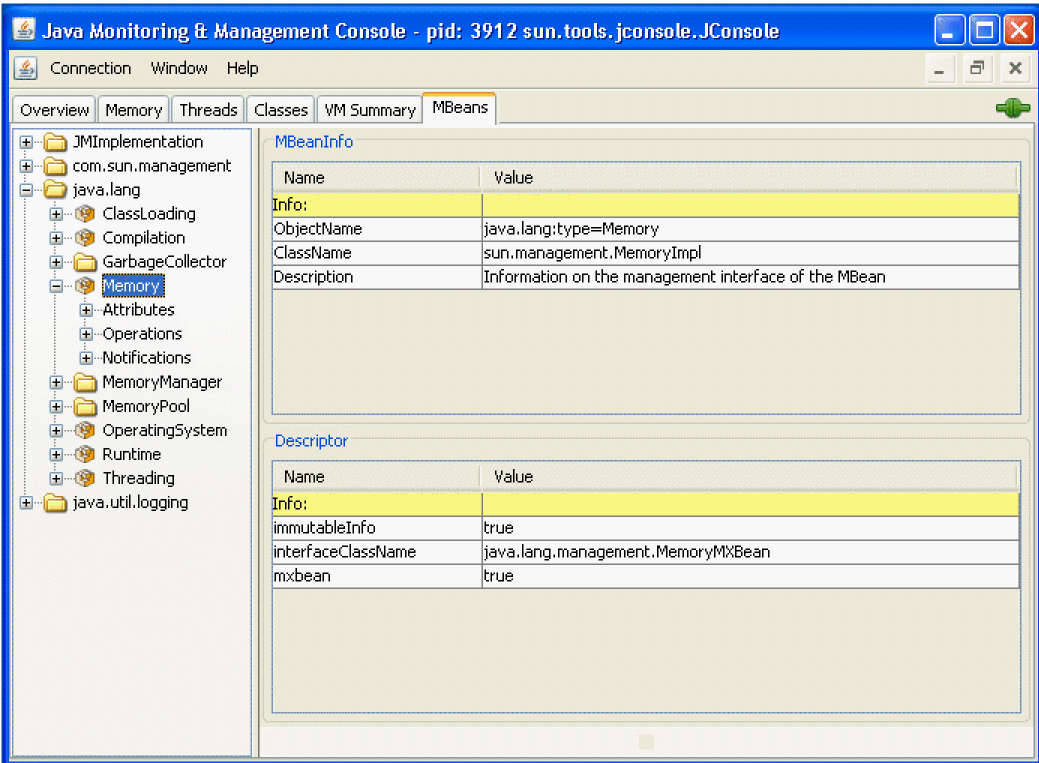
- Total classes loaded: Total number of classes loaded into memory since the Java VM started, including those that have subsequently been unloaded.
- Total classes unloaded: Number of classes unloaded from memory since the Java VM started.

- **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 that 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 path 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 the ones that are 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 from 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.

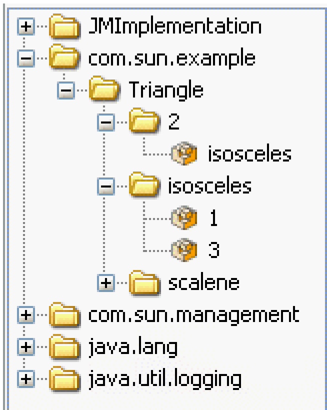
For example, suppose you create `Triangle` MBean objects with the following names.

```
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 subnode 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]*
```

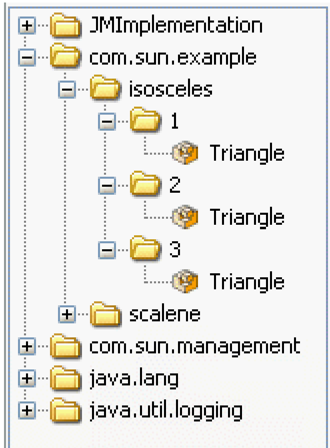
The key property list system property takes a comma-separated list of keys, in the order of your selection, where *key* must be a string representing an object name key or an empty string. If a key specified in the list does not apply to a particular MBean, then that key will be discarded. If an MBean 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 means that JConsole will display keys in the order that they appear in the MBean's ObjectName.

So, returning to the example of the Triangle MBeans cited previously, you can start JConsole by 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, start the JConsole with the following command:

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

Starting JConsole with this system property specified will produce the MBean tree as shown in the Figure 3-14.

Figure 3-14 Example of MBean Tree Constructed Using keyPropertyList



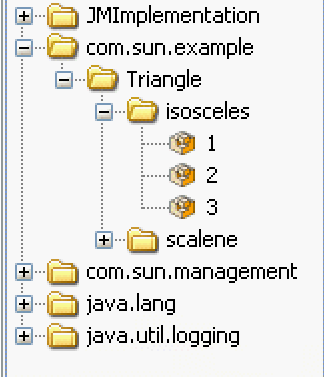
In Figure 3-14, the `side` key comes first, followed by the `name` key. The `type` key comes at the end because it was not specified in the key property list, so the MBean tree algorithm applied the original key order for the remaining keys. Consequently, the `type` key is appended at the end, after 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. You must start JConsole with the following system property:

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

The prior command will cause JConsole to render the MBean tree for the Triangle MBeans as shown in the Figure 3-15.

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

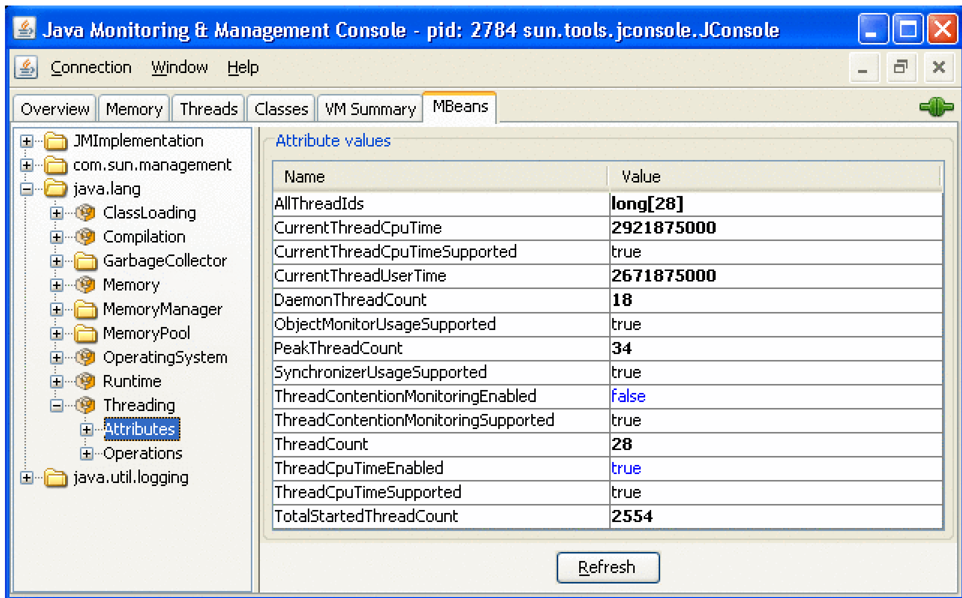


This is comprehensible than the MBean trees as shown in Figure 3-13 and Figure 3-14.

MBean Attributes

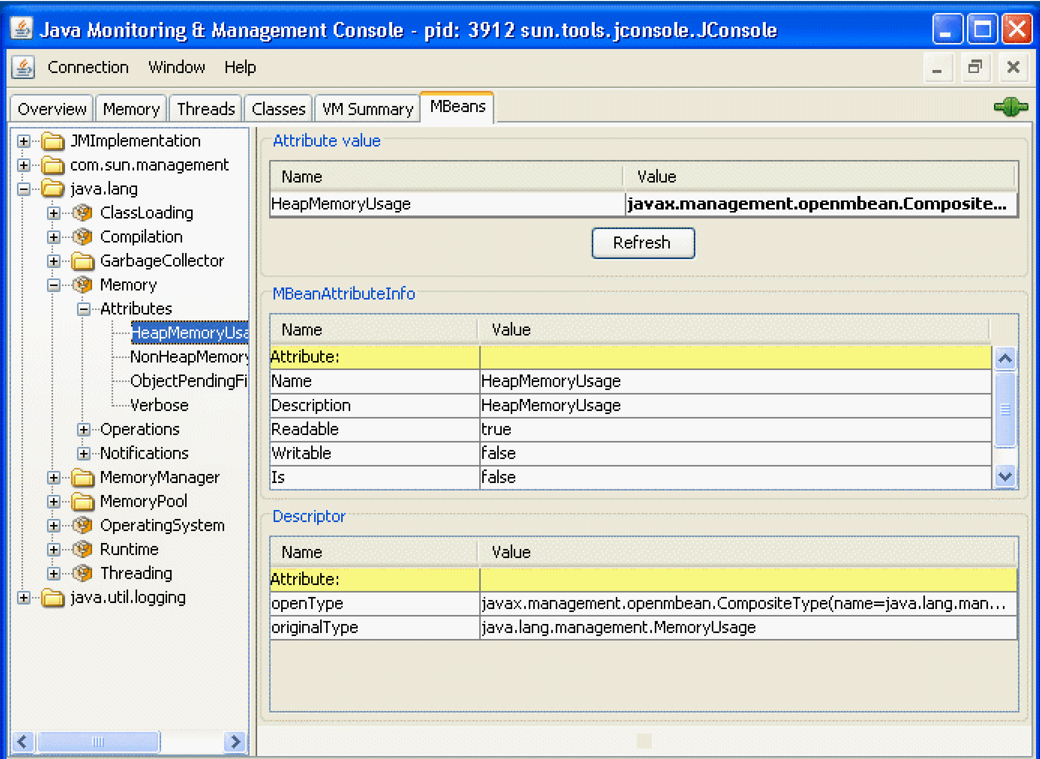
Selecting the **Attributes** node displays all the attributes of an MBean. Figure 3-16 shows all the attributes of the Threading platform MXBean.

Figure 3-16 Viewing All MBean Attributes



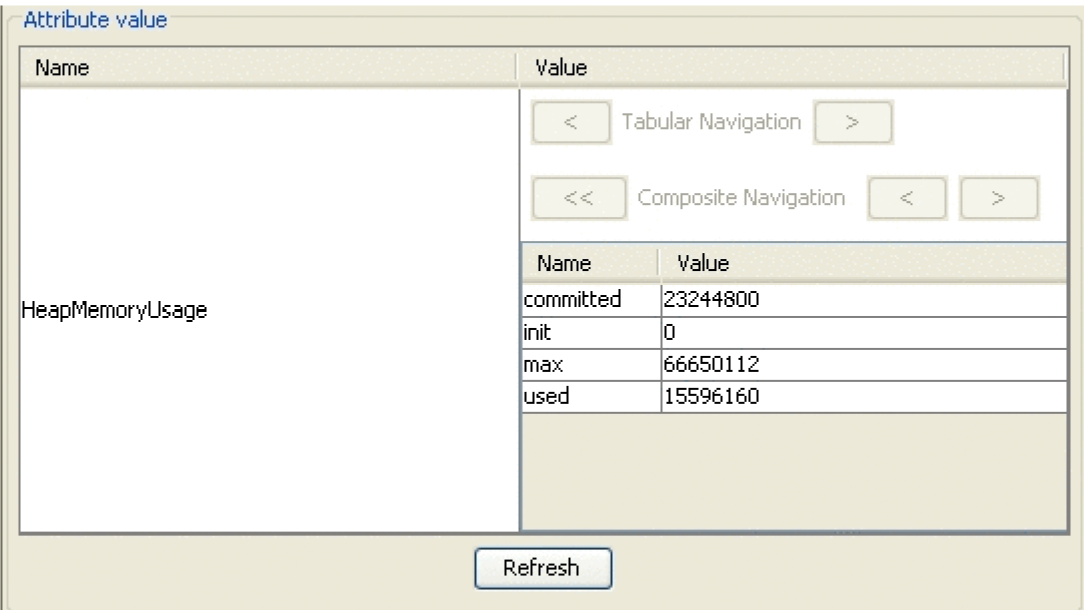
Selecting an individual MBean attribute from the tree then displays the attribute's value, its `MBeanAttributeInfo`, and the associated Descriptor in the right pane, as you can see in Figure 3-17.

Figure 3-17 Viewing an Individual MBean Attribute



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

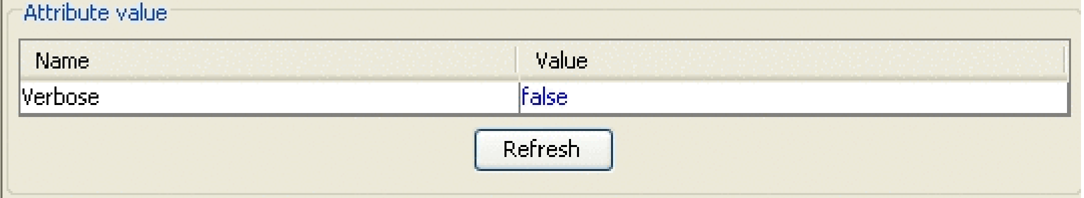
Figure 3-18 Displaying Attribute Values



Double-clicking numeric attribute values will display a chart that plots changes in that numeric value. For example, double-clicking the **CollectionTime** attribute of the Garbage Collector MBean `PS Marksweep` 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

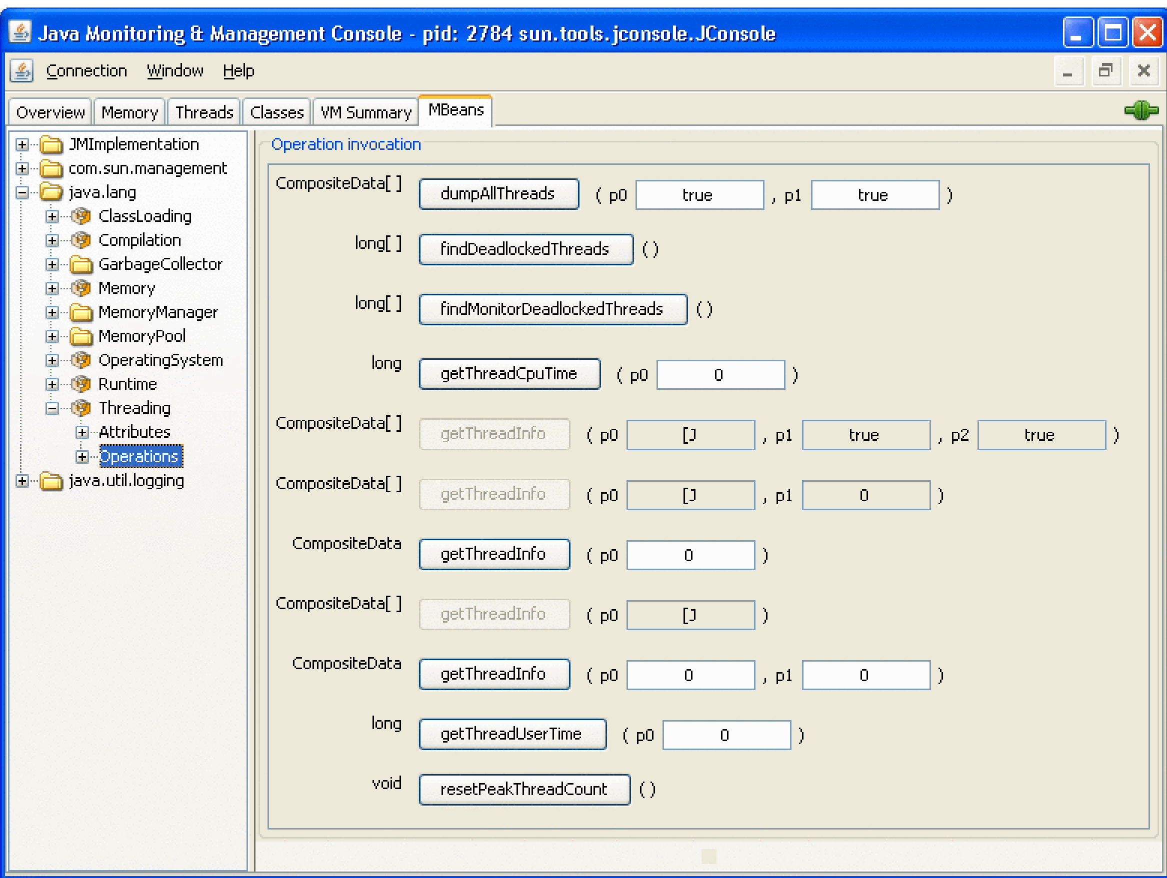


You can set attributes by clicking them and then editing them. For example, to enable or disable the verbose tracing of the garbage collector in JConsole, select the **Memory MXBean** in the **MBeans** tab and set the `Verbose` attribute to `true` or `false`. Similarly, the class loading MXBean also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

MBean Operations

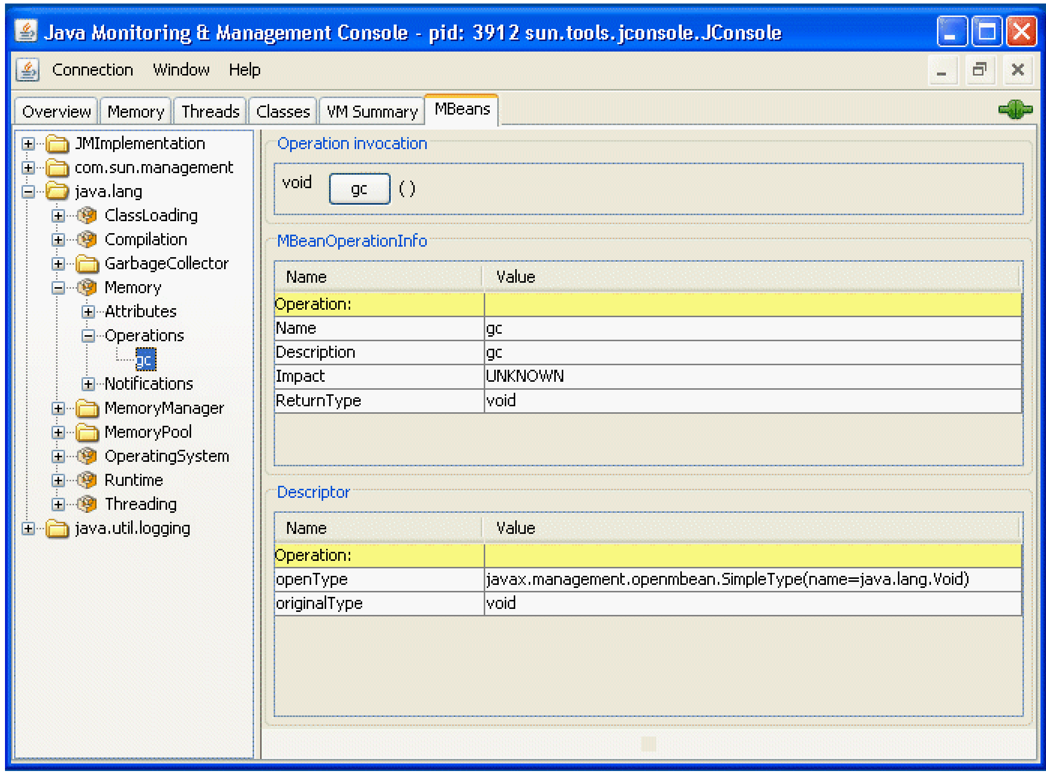
Selecting the **Operations** node displays all the operations of an MBean. The MBean operations appear as buttons, that you can click to call the operation. Figure 3-20 shows all the operations of the Threading platform MXBean.

Figure 3-20 Viewing All MBean Operations



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

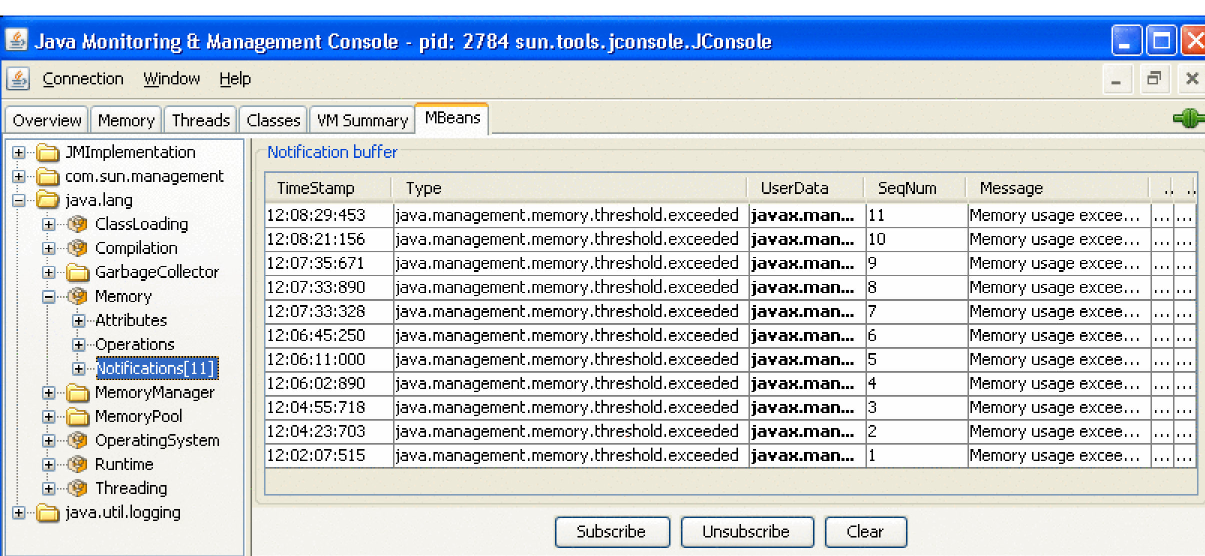
Figure 3-21 Viewing Individual MBean Operations



MBean Notifications

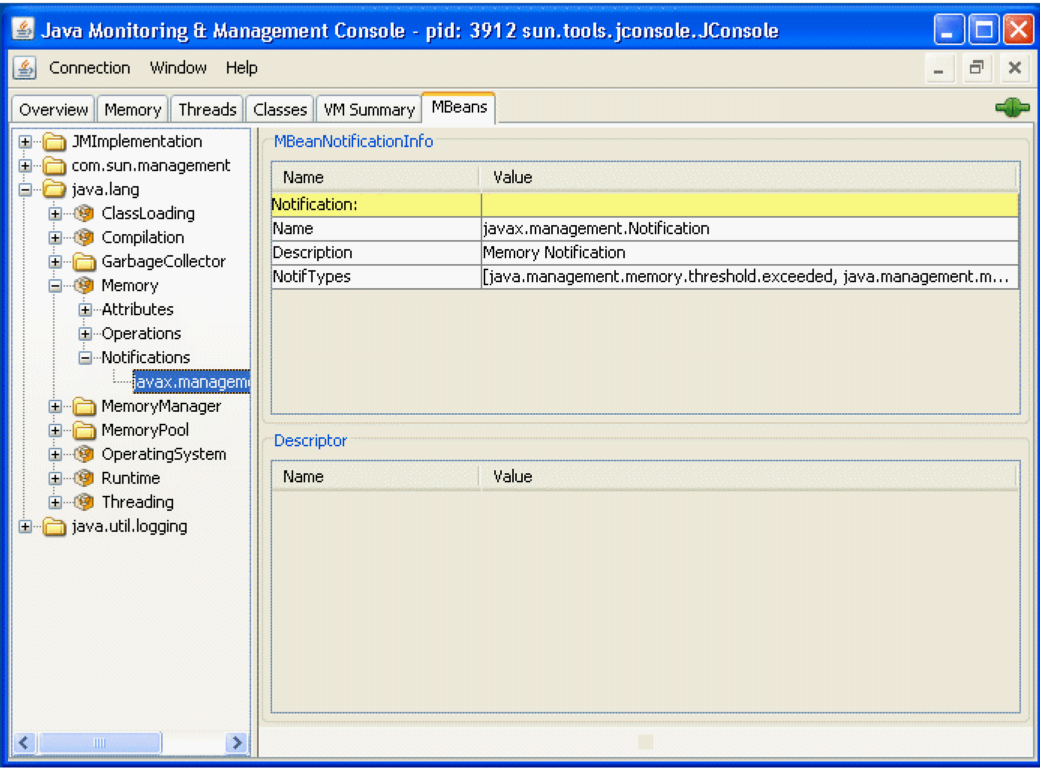
You can subscribe to receive notifications by selecting the **Notifications** node in the left-hand tree, and clicking the **Subscribe** button that appears on the right. The number of notifications received is displayed in brackets, and the Notifications node itself will appear in bold text when new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

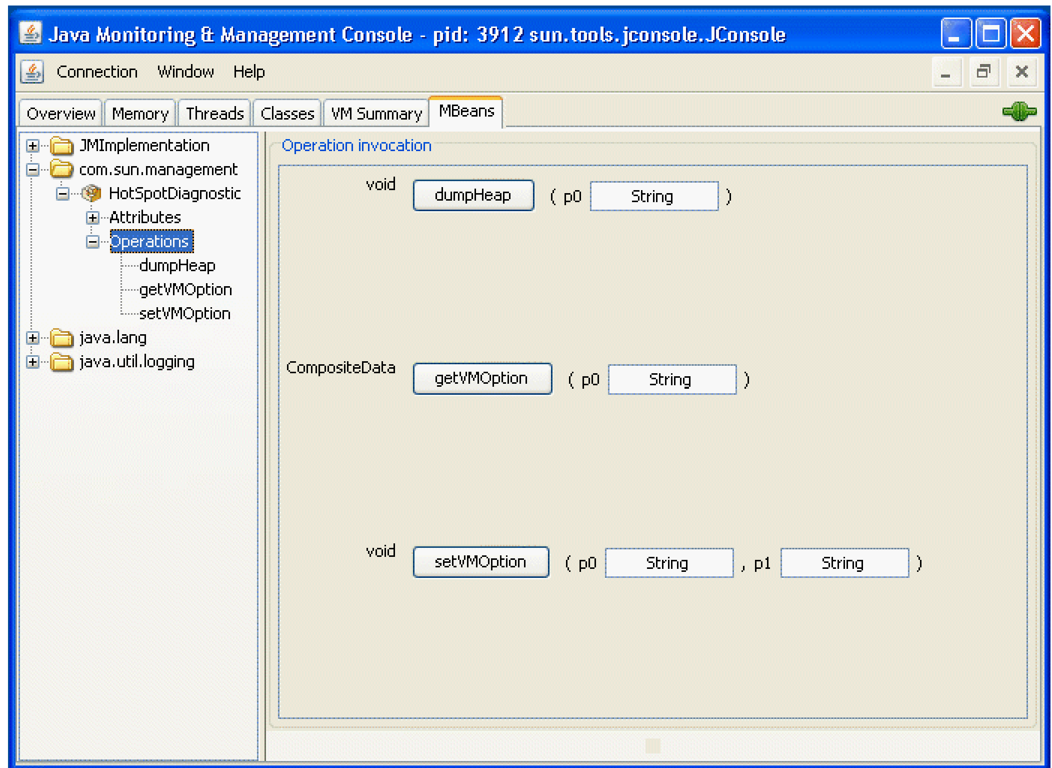
Figure 3-23 Viewing Individual MBean Notifications



HotSpot Diagnostic MXBean

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 using the `HotSpotDiagnostic` MXBean.

Figure 3-24 Viewing the HotSpot Diagnostic MBean



You can perform a heap dump manually by calling the `com.sun.management.HotSpotDiagnostic` MXBean'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 previously, your plug-in must extend `JConsolePlugin`, and implement the `JConsolePlugingetTabs` 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 that 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
```

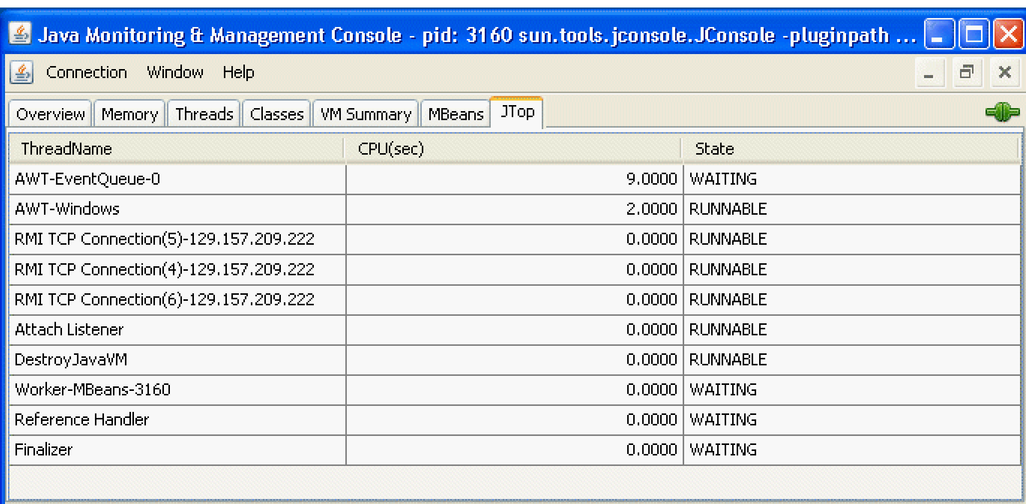
In the previous command, *plugin-path* specifies the paths to the JConsole plug-ins to be looked up. These paths can either be to directory names or to JAR files, and multiple paths can be specified, using your platform's standard separator character.

An example JConsole plug-in is provided with the Java SE 12 platform. The `JTop` application is a JDK demonstration (demo) 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 12 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, then you will see that the `JTop` tab has been added, showing CPU usage of the various threads running.

Figure 3-25 Viewing a Custom Plug-in Tab



4 Using the Platform MBean Server and Platform MXBeans

This topic introduces the MBean server and the MXBeans that are provided as part of the Java Platform, Standard Edition (Java SE), which can be used for monitoring and management purposes. Java Management Extensions (JMX) technology MBeans and MBean servers were introduced briefly in Overview of Java SE Monitoring and Management. See Introduction to JMX Technology in *Java Platform, Standard Edition Java Management Extensions Guide*.

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 using their unique `ObjectName` class. An MBean server implements the interface `javax.management.MBeanServer`.

The *platform MBean server* was introduced in Java SE 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 - Platform MXBeans.

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

- Direct access from the `ManagementFactory` class
- Direct access from an MXBean proxy
- Indirect access from the `MBeanServerConnection` class

Accessing Platform MXBeans Using 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. The `ManagementFactory` class has `accessor` methods for each of the different platform MXBeans, such as, `getClassLoadingMXBean()`, `getGarbageCollectorMXBeans()`, `getRuntimeMXBean()`,

and so on. In case 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 Using ManagementFactory Class

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

Accessing Platform MXBeans Using an MXBean Proxy

An application can also call platform MXBean methods using 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 it uses an MXBean proxy.

Example 4-2 Accessing a Platform MXBean Using an MXBean Proxy

```
MBeanServerConnection mbs;
...
// Get a MBean proxy for RuntimeMXBean interface
RuntimeMXBean proxy =

ManagementFactory.newPlatformMXBeanProxy(mbs,ManagementFactory.RUNTIME_MXBE
AN_NAME,RuntimeMXBean.class);
// Get standard attribute "VmVendor"
String vendor = proxy.getVmVendor();
```

Accessing Platform MXBeans Using the MBeanServerConnection Class

An application can indirectly call platform MXBean methods through an `MBeanServerConnection` interface that connects to the platform MBean server of another running Java VM. You use the `MBeanServerConnection` class `getAttribute()` method to get an attribute of a platform MXBean by 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 it uses an indirect call through `MBeanServerConnection`.

Example 4-3 Accessing a Platform MXBean Using 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 Oracle JDK's 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 Oracle JDK's platform extensions. The following sections provide examples of how to access a platform-specific attribute from Oracle JDK's implementation of the `OperatingSystemMXBean` interface.

Accessing MXBean Attributes Directly

Example 4-4 illustrates direct access to one of Oracle JDK's MXBean interfaces.

Example 4-4 Accessing an MXBean Attribute Directly

```
com.sun.management.OperatingSystemMXBean mxbean =
  (com.sun.management.OperatingSystemMXBean)
ManagementFactory.getOperatingSystemMXBean();

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

// Get the Oracle JDK-specific attribute Process CPU time
long cpuTime = mxbean.getProcessCpuTime();
```

Accessing MXBean Attributes Using MBeanServerConnection

Example 4-5 illustrates access to one of Oracle JDK's MXBean interfaces using the `MBeanServerConnection` class.

Example 4-5 Accessing an MXBean Attribute Using MBeanServerConnection

```
MBeanServerConnection mbs;

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

try {
    // Assuming the OperatingSystem MXBean 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 MXBean contains Oracle JDK's extension
    if (mbs.isInstanceOf(oname,
"com.sun.management.OperatingSystemMXBean")) {
        // 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 Oracle JDK's 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 Oracle JDK's 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 Solaris, Linux, or macOS platforms).

When the Operating System 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 MXBean for logging purposes, the `LoggingMXBean` interface.

The `LoggingMXBean` interface 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 `LoggingMXBean` is `java.util.logging:type=Logging`. This object name is stored in the `LogManager.LOGGING_MXBEAN_NAME` field.

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

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

The `Logging MXBean` interface 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 that 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 that 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*.

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, because 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), 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 because 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 the `MemoryPoolMXBean` interface 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 using 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 nonheap 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, then 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;
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 the `getUsageThresholdCount()` method 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 shown in the Example 4-6.

Threshold Notifications

When the `MemoryMXBean` interface detects that a memory pool has reached or exceeded its usage threshold, it emits a *usage threshold exceeded* notification. The `MemoryMXBean` interface 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` interface 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. As multiple threads can concurrently call the notification listener, the listener should synchronize the tasks that 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` class will handle notification.

5 Java Discovery Protocol (JDP)

The Java Discovery Protocol (JDP) is a protocol that enables technologies, in particular, Java Mission Control and Java Flight Recorder, to discover manageable JVMs across the same network subnet.

A manageable JVM is one that has the Java Management Extensions (JMX) agent running. JDP is multicast-based and works like a beacon; it broadcasts the JMX service URL (see the class JMXServiceURL) required to connect to the external JMX agent. This enables technologies to detect JVMs that have failed or are no longer available for monitoring.

Enabling and Configuring JDP

To enable JDP, specify the following option at the command line when starting a Java application:

```
-Dcom.sun.management.jmxremote.autodiscovery=true
```

Note:

Enabling JDP does not affect JMX security. To enable and configure JMX security, see Monitoring and Management Using JMX Technology.

Table 6-1 describes other properties that you may set to configure JDP:

Table 5-1 JDP Properties

Property	Description	Default Value
`-Dcom.sun.management.autodiscovery`	Enables autodiscovery (JDP) on the network subnet	false
`-Dcom.sun.management.jdp.pause`	Specifies the broadcast interval in seconds	5
`-Dcom.sun.management.jdp.ttl`	Time-to-live in seconds for autodiscovery packets	1
`-Dcom.sun.management.jdp.address`	Multicast address to send autodiscovery packets	224.0.23.178

Table 5-1 (Cont.) JDP Properties

Property	Description	Default Value
-Dcom.sun.management.jdp.port	Multicast port to send autodiscovery packets. Enables autodiscovery even if the `com.sun.management.autodiscovery` property has not been set.	7095
-Dcom.sun.management.jdp.name	Broadcast name of the JVM	No default
-Dcom.sun.management.jdp.source_addr	Address of source interface to use for broadcast	Automatically assigned