WebSphere®

IBM WebSphere Real Time V2 for RT Linux

Version 2

User Guide

IBM®

WebSphere®

IBM WebSphere Real Time V2 for RT Linux

Version 2

User Guide

IBM®

Note
Before using this information and the product it supports, read the information in "Notices" on page 477.

Sixth Edition (April 2010)

This edition of the user guide applies to IBM WebSphere Real Time, Version 2, and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

Preface

This user guide provides general information about IBM® WebSphere® Real Time for RT Linux®.

Chapter 1. Introduction

This information center describes the IBM WebSphere Real Time for RT Linux product referred to as WebSphere Real Time for RT Linux in this information.

You can use this information to install and configure WebSphere Real Time for RT Linux. Selected information about non-Real-Time Java™ is also provided here. See the Diagnostics Guide for further diagnostic information.

The included sample application shows how a standard Java application can take advantage of real-time capabilities to improve predictability.

For the latest information see the `readmeFirst.txt` file in the docs directory.

Viewing the information center

The documentation for real-time Java is in `com.ibm.rt.doc.20_3.0.jar`, `com.ibm.rt.doc.20_3.0.zip`, and `rt_jre.pdf` files are in the `docs` directory.

- You can copy `com.ibm.rt.doc.20_3.0.jar` to either the `plug-ins` directory of your Eclipse Help System version 3.1.1 or to the `plug-ins` directory of Eclipse SDK version 3.1.2 or later.
- You can unpack `com.ibm.rt.doc.20_3.0.zip` into the `plug-ins` directory of your Eclipse Help System if the version is earlier than 3.1.1.
- The `rt_jre.pdf` file uses Adobe® Acrobat.

To use the information center on your local workstation, you must install the Eclipse Help system; see "Viewing the online help" on page 14. You can also copy the jar file to the `plug-in` directory of Eclipse SDK and view the information center using **Help → Help Contents**.

Who should read this information

Readers of this information fall into one of the following groups:

- Java application programmers who develop WebSphere Real Time for RT Linux applications; see Table 1.
- System administrators who install and configure the Java environment; see Table 2 on page 2.
- Service personnel team who maintain the Java environment; see Table 3 on page 2.

Table 1. Application programmers' tasks

Task	Reference
Planning your application.	"Planning your WebSphere Real Time for RT Linux application" on page 88
Running a standard Java application in real time without any changes to the process or the code.	"Running the sample application with Metronome Garbage Collector" on page 79

Table 1. Application programmers' tasks (continued)

Task	Reference
Running a standard Java application in real time but avoiding the unpredictable time delays caused by JIT.	"Running the sample application using AOT" on page 80 Chapter 4, "Using compiled code with WebSphere Real Time for RT Linux," on page 23
Writing a Java application that has a more predictable time control.	"Writing real-time threads" on page 90
Writing a Java application that can respond to external events.	"Writing asynchronous event handlers" on page 93
Writing Java application components with sub-millisecond precision that cannot tolerate any delay caused by garbage collection activities.	"Writing NHRT threads" on page 94

Table 2. System administrators' tasks

Task	Reference
Planning for and overseeing product installation.	Chapter 2, "Installing WebSphere Real Time for RT Linux," on page 9
Tuning the real-time environment.	"Tuning Metronome Garbage Collector" on page 58

Table 3. Service personnel tasks

Task	Reference
Troubleshooting system and performance problems.	Chapter 12, "Troubleshooting OutOfMemory Errors," on page 109
Diagnosing problems.	"First steps in problem determination" on page 123

Overview of WebSphere Real Time for RT Linux

WebSphere Real Time for RT Linux bundles real-time capabilities with the standard JVM.

You enable real-time capabilities by using the -Xrealtime option when running the JVM or any of the tools provided. By default, the JVM and the tools provided run without real-time capabilities enabled. Figure 1 on page 3 shows the relationships of the two JVMs that are supplied with WebSphere Real Time for RT Linux.

The following Java commands recognize the -Xrealtime option:

Table 4. Java commands used in real-time mode

Command	Function
java	Runs in standard mode by default but also runs in real-time mode when the -Xrealtime option is specified. In real-time mode, the programmer accesses classes from the javax.realtime package. You can use precompiled jar files from the jxeinajar tool and the Metronome deterministic garbage collection technology.
javac, javah, javap	Runs in standard mode by default; but, when the -Xrealtime option is specified, it includes the javax.realtime.* classes in the classpath.

Table 4. Java commands used in real-time mode (continued)

Command	Function
jxeinajar	Runs in real-time mode only. Running jxeinajar without the -Xrealtime option is not supported. jxeinajar eases migration for WRT V1 users who used the tool to AOT compile their code. It is not meant for production scenarios. From WebSphere Real Time for RT Linux SR1 onwards, the jxeinajar tool is not supported. Instead, you must use the admincache tool. See "WebSphere Real Time V1 Migration" on page 38 for information on how to do this.
admincache	Can be run both with and without -Xrealtime, but populating a shared cache using the admincache tool is only possible in the real-time mode. In regular mode, only the cache utilities are available (listAllCaches, printStats, and so on.). Like jdmpview, admincache must be run with -Xrealtime to access caches for the real-time JVM, and must be run without -Xrealtime to access caches for the regular JVM. From WebSphere Real Time for RT Linux SR1 onwards, it is mandatory that you use the **-classpath** with the **-populate** option.
jextract	jextract runs in standard mode by default, but must be run with the -Xrealtime option when processing system dumps generated by the JVM in real-time mode

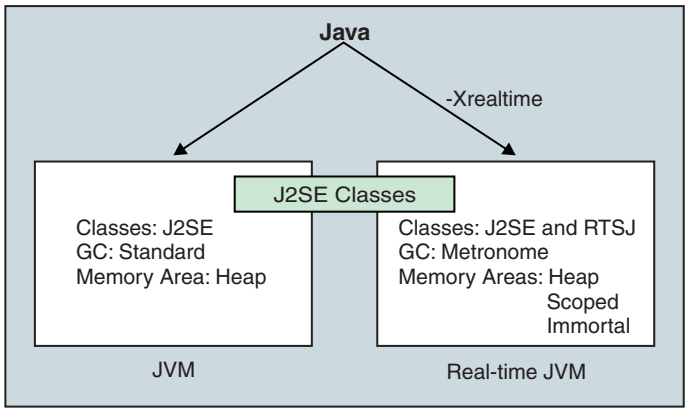


Figure 1. Overview of WebSphere Real Time for RT Linux

Features of WebSphere Real Time for RT Linux

Real-time applications need consistent run time rather than absolute speed.

When the JVM is run with the -Xrealtime option, additional memory areas are available in addition to the garbage collected heap. Programs might request or specify any number of reusable scoped and nonreusable immortal memory areas, which are not garbage collected, providing the application with more control over memory usage. It also uses the Metronome Garbage Collector to achieve time-based collections. When the JVM is run in a traditional throughput mode without the -Xrealtime option, various work-based garbage collectors can be used that optimize throughput but can have larger individual delays than the Metronome Garbage Collector.

The main concerns when deploying real-time applications with traditional JVMs are as follows:
- Unpredictable (potentially long) delays from Garbage Collection (GC) activity.
- Delays to method runtime as Just-In-Time (JIT) compilation and recompliation occurs, with variability in execution time.
- Arbitrary operating system scheduling.

WebSphere Real Time for RT Linux removes these obstacles by providing:
- The Metronome Garbage Collector, an incremental, deterministic garbage collector with very small pause times
- Ahead-Of-Time (AOT) compilation
- Priority based FIFO scheduling

In addition, WebSphere Real Time provides the real-time programmer with the RTSJ facilities; see Chapter 7, "Support for RTSJ," on page 61.

What's new

This topic introduces anything new for IBM WebSphere Real Time for RT Linux Refreshes

What's new for WebSphere Real Time for RT Linux Refresh 3

- Java thread names are visible in the operating system when using the ps command. For further information about using the ps command, see "Examining process information" on page 128.
- Regular Java threads can be scheduled using the SCHED_RR or SCHED_FIFO scheduling policies. For further information, see Chapter 3, "Thread scheduling and dispatching," on page 17.

What's new for WebSphere Real Time for RT Linux Refresh 2

- Security considerations for the shared class cache.

The shared class cache is designed for ease of cache management and usability, but the default security policy might not be appropriate.

When using the shared class cache, you must be aware of the default permissions for new files so that you can improve security by restricting access.

File	Default permissions
new shared caches	read permissions for group and other
`javasharedresources` directory	world read, write, and execute permission

You require write permission on both the cache file and the cache directory to destroy or grow a cache.

For information about how to change the permissions on a shared cache or directory, see "Security considerations for the shared class cache" on page 7.

What's new for WebSphere Real Time for RT Linux Refresh 1

- There are no significant updates.

Benefits

The benefits of the real-time environment are that Java applications run with a greater degree of predictability than with the standard JVM and provide consistent timing behavior for your Java application. Background activities, such as compilation and garbage collection, occur at given times and thus remove any unexpected peaks of background activity when running your application.

You obtain these advantages by extending the JVM with the following functions:

- Metronome real-time garbage collection technology
- Ahead-of-time (AOT) compilation
- Support for the Real-Time Specification for Java (RTSJ)

All Java applications can be run in a real-time environment without modification, benefiting from the Metronome Garbage Collector and its deterministic garbage collection that occurs at regular intervals. To achieve the maximum benefit from WebSphere Real Time for RT Linux, you can write applications specifically for the real-time environment using both real-time threads and no-heap real-time threads. The approach that you take depends on the timing specification of your application.

Many real-time Java applications can exploit the low pause times of the Metronome Garbage Collector and AOT to achieve their goals, retaining the benefits of Java portability. Applications with tighter requirements must use the RTSJ facilities of real-time threads and no-heap real-time threads, with scoped and immortal memory. This approach limits your application to run in a real-time environment only, losing the advantage of portability to JSE Java. You also have to develop a more complex programming model.

Considerations

You must be aware of a number of factors when using WebSphere Real Time for RT Linux.

- Where possible, do not run more than one real-time JVM on the same system. The reason is that you would then have multiple garbage collectors. Each JVM does not know about the memory areas of the other. Therefore, neither JVM can know what is feasible.
- You cannot use the **-Xdebug** option and the **-Xnojit** option with code that has been precompiled using the Ahead-of-Time compiler. The reason is that **-Xdebug** compiles code in a different way from the Ahead-of-Time (AOT) compiler and is not supported.

  To debug your code, use interpreted or JIT-compiled code.
- If you are using the com.sun.tools.javac.Main interface to compile Java source code that uses the javax.realtime package, you must ensure that `sdk/jre/lib/i386/realtime/jclSC160/realtime.jar` is included in the class path. One common example of this type of compilation is ant compilation.
- The optional JavaComm package can be installed into WebSphere Real Time for RT Linux and accessed from both the real-time and non-real-time JVM. For more information about installation and configuration, see “Java Communications API (JavaComm)” on page 407. The real-time JVM in WRT supports the JavaComm API for use with regular Java threads. However, no guarantee exists in respect to determinism or real-time performance when accessing external devices with JavaComm. As such, do not use JavaComm with no-heap real-time and real-time threads, or when real-time behavior is required.

- When using shared class caches, the cache name must not exceed 53 characters.
- The ps command truncates Java thread names.

  The ps command is limited to 15 characters. If you set a thread name to more than 15 characters, the name is truncated by the ps command.
- Changed thread names.

  Some internal JVM thread names have changed in WebSphere Real Time for RT Linux 2 SR 3. For example, the default name for a real-time thread is `RTThread-n`, where `n` is an integer to identify the exact threat. Similarly, the default name for a no-heap real-time thread is `NHRTThread-n`.
- WebSphere Real Time for RT Linux does not support NTLoginModule (NTLM) authentication.

  NTLoginModule (NTLM) is used to help authenticate access to a Windows® service. Authentication using NTLM is supported on the Windows platform only. This means that WebSphere Real Time for RT Linux does not support NTLM authentication.

Performance considerations

WebSphere Real Time for RT Linux is optimized for consistently short GC pauses rather than the highest throughput performance or smallest memory footprint.

On systems where hyperthreading is supported, you must ensure that it is not enabled. This is to avoid adverse performance effects when using WebSphere Real Time for RT Linux.

Reducing timing variability and support for the Real-Time Specification for Java (RTSJ) required some standard IBM Java runtime optimizations to be disabled. Consequently, a reduction in overall performance is likely to be observed when a standard Java application is run with the **-Xrealtime** parameter.

Performance on certified hardware configurations

Certified systems have sufficient clock granularity and processor speed to support WebSphere Real Time for RT Linux performance goals. For example, a well written application running on a system that is not overloaded, and with an adequate heap size, would normally experience GC pause times that are well below 1 millisecond, typically about 500 microseconds. During GC cycles, an application with default environment settings is not paused for more than 30% of elapsed time during any sliding 10 millisecond window. The collective time spent in GC pauses over any 10 millisecond period should add up to less than 3 milliseconds.

Reducing timing variability

The two main sources of variability in a standard JVM are handled in WebSphere Real Time for RT Linux as follows:

- Java code preparation: loading and Just-In-Time (JIT) compilation is dealt with by Ahead-Of-Time (AOT) compilation. See “Using the AOT compiler” on page 27.
- Garbage Collection pauses: the potentially long pauses from standard Garbage Collector modes are avoided by using the Metronome Garbage Collector. See Chapter 6, “Using the Metronome Garbage Collector,” on page 49.

Scheduling policies and priorities

From V2 SR3, regular Java threads can run with the policy SCHED_FIFO in addition to the default policy SCHED_OTHER. When running with the policy SCHED_FIFO, threads can run with a Linux priority 1 - 10, giving you finer control over your application. For more information about thread scheduling and dispatching, see Chapter 3, "Thread scheduling and dispatching," on page 17.

Related concepts

"Launching secondary processes" on page 20
The java.lang.Runtime.exec methods in the Java virtual machine (JVM) API give your Java application the ability to execute a command in a separate process.

Class data sharing between JVMs for non Real-Time

Class sharing is supported in non-real-time mode, but operates differently than in real-time mode.

The Java Virtual Machine (JVM) allows you to share class data between JVMs by storing it in a memory-mapped cache file on disk. Sharing reduces the overall virtual storage consumption when more than one JVM shares a cache. Sharing also reduces the startup time for a JVM after the cache has been created. The shared class cache is independent of any running JVM and persists until it is destroyed.

A shared cache can contain:

- Bootstrap classes
- Application classes
- Metadata that describes the classes
- Ahead-of-time (AOT) compiled code

Security considerations for the shared class cache

The shared class cache is designed for ease of cache management and usability, but the default security policy might not be appropriate.

When using the shared class cache, you must be aware of the default permissions for new files so that you can improve security by restricting access.

File	Default permissions
new shared caches	read permissions for group and other
`javasharedresources` directory	world read, write, and execute permission

You require write permission on both the cache file and the cache directory to destroy or grow a cache.

Changing the file permissions on the cache file

To limit access to a shared class cache, you can use the **chmod** command.

Change required	Command
Limit access to the user and group	`chmod 770 /tmp/javasharedresources`
Limit access to the user	`chmod 700 /tmp/javasharedresources`
Limit the user to read and write access only for a particular cache	`chmod 600 /tmp/javasharedresources/<file for shared cache>`

Change required	Command
Limit the user and group to read and write access only for a particular cache	`chmod 660 /tmp/javasharedresources/<file for shared cache>`

See “Creating a Real-Time Shared Class Cache” on page 29 for more information about creating a shared class cache.

Connecting to a cache that you do not have permission to access

If you try to connect to a cache that you do not have the appropriate access permissions for, you see an error message:

```
JVMSHRC226E Error opening shared class cache file
JVMSHRC220E Port layer error code = -302
JVMSHRC221E Platform error message: Permission denied
JVMJ9VM015W Initialization error for library j9shr25(11): JVMJ9VM009E J9VMDllMain failed
Could not create the Java virtual machine.
```

Related concepts

“Cache access” on page 253
A JVM can access a shared class cache with either read-write or read-only access. Read-write access is the default and allows all users equal rights to update the cache. Use the **-Xshareclasses:readonly** option for read-only access.

Migration

WebSphere Real Time for RT Linux runs in a Linux environment modified for real-time applications. You can use standard Java applications in a real-time environment. Alternatively, you can modify your applications to exploit the new features of WebSphere Real Time.

WebSphere Real Time for Real Time Linux is fully supported on Red Hat Enterprise Linux MRG Version 1 or SUSE Linux Enterprise Real Time 10. These are Linux distributions modified with open source, real-time scripts.

System migration

Follow the instructions provided by the Linux support team.

Chapter 2. Installing WebSphere Real Time for RT Linux

Follow these steps to install the product.

Installation files

You require these installation files.

- The installation file for IBM WebSphere Real Time for RT Linux is called `ibm-wrt-sdk-2.0-2.0-linux-i386.tgz`.
- The IBM Eclipse Help system, Version 3.1.1. can be installed locally and the WebSphere Real Time for RT Linux documentation plug-in jar file can be copied to the plug-in directory. See http://www.alphaworks.ibm.com/tech/iehs/download.

Hardware and software prerequisites

Use this list to check the hardware, operating system, and Java environment that is supported for WebSphere Real Time for RT Linux.

Hardware

WebSphere Real Time for RT Linux certified hardware configurations are multiprocessor variants of the following systems:

- IBM BladeCenter® LS20 (Types 8850-76U, 8850-55U, 7971, 7972)
- IBM eServer™ xSeries® 326m (Types 7969-65U, 7969-85U, 7984-52U, 7984-6AU)
- IBM BladeCenter LS21 (Type 7971-6AU)
- IBM BladeCenter HS21 XM Dual Quad Core (Type 7995)

To remain certified for WebSphere Real Time for RT Linux, IBM systems with hyperthreading must not have hyperthreading enabled.

In addition, WebSphere Real Time for RT Linux is supported on hardware that runs a supported operating system, and that has these characteristics:

- A minimum of 512 MB of physical memory.
- AMD Opteron, Intel® Pentium® processor, or Intel EM64T running at 800 MHz or faster.
- WebSphere Real Time for RT Linux for 32-bit: minimum Intel Pentium 3, AMD Opteron, or Intel Atom Processor.

For systems that are not certified hardware configurations, IBM does not make any performance statements. Performance considerations for certified hardware configurations are detailed here: “Performance considerations” on page 6

On systems with hyperthreading support, ensure that it is not enabled to avoid adverse performance effects when using WebSphere Real Time for RT Linux.

Operating system

- Red Hat Enterprise Linux 5.3 MRG. See “Installing a Real Time Linux environment.”
- SUSE Linux Enterprise Real Time 10. See “Installing a Real Time Linux environment.”

Related information

“Setting up and checking your Linux environment” on page 125
Linux operating systems undergo a large number of patches and updates.

Useful tools

You can use these tools with WebSphere Real Time for RT Linux. Some of these tools are in the early stages of development and might not be fully supported.

For application development, Eclipse SDK 3.1.2 or later provides a complete application development environment for real-time applications. You can also copy the information center file, `com.ibm.rt.doc.20_3.0.jar`, to the plug-in directory to view the documentation associated with this product. This product can be downloaded from http://www.eclipse.org/downloads/.

You can find information about the latest tools that you can use to support WebSphere Real Time for RT Linux; for example: Tuning fork, eventrons, and real-time class analysis tool (ratcat). These tools can be downloaded from http://www.alphaworks.ibm.com/keywords/Real-time%20Java.

Related tasks

Chapter 16, “Developing WebSphere Real Time for RT Linux applications using Eclipse,” on page 347
Using Eclipse provides you with a fully-featured IDE when developing your real-time applications.

Installing a Real Time Linux environment

Before you can install WebSphere Real Time for RT Linux, you must install Real Time Linux.

Before you begin

Before installing WebSphere Real Time for RT Linux, you must install a 64-bit version of Real Time Linux.

Red Hat Enterprise Linux 5.3 MRG

- For more information about installing the real time component of Red Hat Enterprise Linux 5.3 MRG, see the installation instructions for RT-Linux RHEL 5.3 MRG 1.1.2: https://www.redhat.com/docs/en-US/Red_Hat_Enterprise_MRG/1.1/html/Realtime_Installation_Guide/index.html

SUSE Linux Enterprise Real Time 10

- For more information about installing SUSE Linux Enterprise Real Time 10, see: http://www.novell.com/products/realtime/eval.html

When using a large number of file descriptors to load different instances of classes, you might see an error message "`java.util.zip.ZipException: error in opening zip file`", or some other form of IOException advising that a file could not be opened. The solution is to increase the provision for file descriptors, using the ulimit command. To find the current limit for open files, use the command:

```
ulimit -a
```

To allow more open files, use the command:

```
ulimit -n 8196
```

Related information

"Setting up and checking your Linux environment" on page 125
Linux operating systems undergo a large number of patches and updates.

Unpacking the WebSphere Real Time for RT Linux gzipped tar file

The JVM is supplied in a gzipped file called `ibm-srt-sdk-2.0-2.0-linux-i386.tgz`. It can be installed into any directory.

About this task

To unpack the Java driver, follow these instructions.

Procedure

1. From a shell prompt, enter `useradd -G realtime -m` *`username`*. This command adds the *username* to the realtime group. Without this access you lack the capabilities to do things like request SCHED_FIFO locked memory. If you are not a member of the realtime group, the following message is issued:

   ```
   rtcheck failed with exit code 1.
   This JVM must run on a specialized real-time version of Linux (TM).
   In addition the user must have the credentials to use real-time capabilities.
   Running:
   rtcheck -v
   will help you find out why the JVM failed.
   ```

   To add a user to the realtime group, substitute your login name for *username*. To make this substitution, you need superuser authority.

2. From a shell prompt, enter:

   ```
   tar xzf ibm-wrt-sdk-2.0-2.0-linux-i386.tgz -C target_directory
   ```

   where *target_directory* is your working directory. This command creates the following directories:

   ```
   ibm-wrt-i386-60/
                       bin/
                       copyright
                       demo/
                       docs/
                       include/
                       jre/
                       lib/
                       license_en.html
                       readmeFirst.txt
                       realtime.src.jar
                       sample/
                       src.zip
   ```

What to do next

See “Setting the path” to set your PATH environment variable.

Setting the path

When you have set the **PATH** environment variable, you can run an application or program by typing its name at a shell prompt.

About this task

Note: If you alter the **PATH** environment variable as described in this section, you override any existing Java executables in your path.

You can specify the path to a tool by typing the path before the name of the tool each time. For example, if the SDK is installed in `/opt/ibm/ibm-wrt-i386-60/`, you can compile a file named *myfile.java* by typing the following at a shell prompt:

```
/opt/ibm/ibm-wrt-i386-60/bin/javac myfile.java
```

To avoid typing the full path each time:

1. Edit the shell startup file in your home directory (usually .bashrc, depending on your shell) and add the absolute paths to the **PATH** environment variable; for example:

   ```
   export PATH=/opt/ibm/ibm-wrt-i386-60/bin:/opt/ibm/ibm-wrt-i386-60/jre/bin:$PATH
   ```

2. Log on again or run the updated shell script to activate the new **PATH** setting.
3. Compile the file with the javac tool. For example, to compile the file *myfile.java*, at a shell prompt, enter:

   ```
   javac -Xrealtime myfile.java
   ```

   The **PATH** environment variable enables Linux to find executable files, such as javac, java, and the javadoc tool, from any current directory. To display the current value of your path, type the following at a command prompt:

   ```
   echo $PATH
   ```

What to do next

See “Setting the classpath” to determine whether you need to set your CLASSPATH environment variable.

Setting the classpath

The **CLASSPATH** environment variable tells the SDK tools, such as java, javac, and javadoc tool, where to find the Java class libraries.

About this task

Set the **CLASSPATH** environment variable explicitly only if one of the following applies:

- You require a different library or class file, such as one that you develop, and it is not in the current directory.
- You change the location of the `bin` and `lib` directories and they no longer have the same parent directory.
- You plan to develop or run applications using different runtime environments on the same system.

To display the current value of your **CLASSPATH**, enter the following at a shell prompt:

```
echo $CLASSPATH
```

If you develop and run applications that use different runtime environments, including other versions that you have installed separately, you must set **CLASSPATH** and **PATH** explicitly for each application. If you run multiple applications simultaneously and use different runtime environments, each application must run in its own shell.

If you run only one version of Java at a time, you can use a shell script to switch between the different runtime environments.

What to do next

See "Testing your installation" to verify that your installation has been successful.

Testing your installation

Use the **-version** option to check if your installation is successful.

About this task

The Java installation consists of a standard JVM and a real-time JVM.

Procedure

Test your installation by following these steps:

1. To see version information for the standard JVM, type the following command at a shell prompt:

   ```
   java -version
   ```

   This command returns the following messages if it is successful:

```
| java version "1.6.0"
| Java(TM) SE Runtime Environment (build pxi3260rtsr3-20100301_01(SR3))
| IBM J9 VM (build 2.4, J2RE 1.6.0 IBM J9 2.4 Linux x86-32 jvmxi3260sr4ifx-20090409_33254 (JIT enabled, AOT enabled)
| J9VM - 20090409_033254_lHdSMr
| JIT  - r9_20090213_2028
| GC   - 20090213_AA)
| JCL  - 20090603_01
```

   If you intend to use the standard JVM and not the real-time JVM, refer to the IBM User Guides for Java v6 on Linux.

   Note: The version information is correct but the dates might be later than the ones in this example. The format of the date string is: `yyyymmdd` followed possibly by additional information specific to the component.

2. To see version information for the real-time JVM, type the following command at a shell prompt:

   ```
   java -Xrealtime -version
   ```

   This command returns the following messages if it is successful:

```
| java version "1.6.0"
| Java(TM) SE Runtime Environment (build pxi3260rtsr3-20100301_01(SR3))
| IBM J9 VM (build 2.5, J2RE 1.6.0 IBM J9 2.5 Linux x86-32 jvmxi32rt60sr3-20100301_36710 (JIT enabled, AOT enabled)
```

```
| J9VM - 20090605_036710
| JIT  - r10_20090603_1712
| GC   - 20090601_AA)
| JCL  - 20090603_01
```

Note: The version information is correct but the dates might be later than the ones in this example. The format of the date string is: `yyyymmdd` followed possibly by additional information specific to the component.

Viewing the online help

In the docs directory, the documentation is provided for use in the Eclipse Help System as `com.ibm.rt.doc.20_3.0.jar` and `com.ibm.rt.doc.20_3.0.zip`. The information is also provided as an Adobe PDF file called `rt_jre.pdf`.

About this task

- `com.ibm.rt.doc.20_3.0.jar` can be copied directly into the `plug-ins` directory of your Eclipse Help System V3.1.1 or the `plug-in` directory of Eclipse SDK V3.1.2 or later.
- `com.ibm.rt.doc.20_3.0.zip` can be unpacked into the `plug-in` directory of your Eclipse Help System if the version is earlier than V3.1.1.
- `rt_jre.pdf` is for use with Adobe Acrobat.

To use the information center on your personal computer, you install the Eclipse Help System.

Note: The information center is also provided as a PDF, but the information has not been fully optimized for this format.

Procedure

1. Install the Eclipse Help System.
   a. Download the latest version of the Eclipse Help System version from http://www.alphaworks.ibm.com/tech/iehs/download.
   b. Select the .zip, .tar, or .tgz file that is appropriate for your operating system.
   c. Create a new directory where you plan to install the Eclipse Help System. This directory is referred to as *<INSTALL_DIR>* in the rest of this document.
   d. Unpack the file into, for example, `/opt/<INSTALL_DIR>` or `C:\<INSTALL_DIR>` directory depending on your operating system. The unpacking creates a directory called `/opt/<INSTALL_DIR>/ibm_help` on Linux or `C:\<INSTALL_DIR>\IBM_Help_301_Win\ibm_help` on Windows.
2. Add the WebSphere Real Time for RT Linux Information Center to your Eclipse Help System.
   - **For Eclipse versions earlier than V3.1.1.** Extract the files from `com.ibm.rt.doc.20_3.0.zip` into the `/opt/<INSTALL_DIR>/ibm_help/eclipse/plugins` directory on Linux or `C:\<INSTALL_DIR>\IBM_Help_301_Win\ibm_help\eclipse\plugins` on Windows.
   - **For Eclipse V3.1.1 or later.** Copy `com.ibm.rt.doc.20_3.0.jar` to the `plug-in` directory in the help system. For example, this directory is `/opt/<INSTALL_DIR>/ibm_help/eclipse/plugins` directory on Linux or `C:\<INSTALL_DIR>\IBM_Help_301_Win\ibm_help\eclipse\plugins` on Windows.
3. Start the Eclipse Help System by changing directory to `/opt/<INSTALL_DIR>/ibm_help` and entering `help_start`.

4. You can use the Eclipse Help System in these ways:
   - Using the search function. The first time you search, the search pauses while indexing takes place.
   - Filtering your searches. You can "Set Scope" so that it searches only the WebSphere Real Time for RT Linux Information Center. Follow the prompts.
   - Printing. From the navigation tree, click the icon that appears when you hover over a topic in the navigation tree. Use the pop-up menu to select that topic or all of the subtopics associated with that topic. Click your preference and a new window opens for you to confirm that you want to print that part. Submit the job to your local printer in the typical way.
   - Installing on a Local Area Network. See the release notes that come with the Eclipse Help System for more information.
   - Using a CD. See the release notes that come with the Eclipse Help System for more information.
5. Close the Eclipse Help System. When you have finished with the help system, enter `help_end`. Otherwise, the next time you try to start the system, you will not be able to start it because of a running process.

Uninstalling IBM WebSphere Real Time for RT Linux

Follow these steps to remove the WebSphere Real Time for RT Linux, version 2.0 that was extracted from the compressed package.

Procedure

1. Remove the SDK or Runtime Environment files from the directory in which you installed the SDK or Runtime Environment.
2. Remove from your **PATH** statement the directory in which you installed the SDK or Runtime Environment.
3. Log on again or run the updated shell script to activate the new **PATH** setting.

Chapter 3. Thread scheduling and dispatching

The Linux operating system supports various scheduling policies. The default universal time sharing scheduling policy is SCHED_OTHER, which is used by most threads. SCHED_RR and SCHED_FIFO can be used by threads in real-time applications. .

The kernel decides which is the next runnable thread to be run by the CPU. The kernel maintains a list of runnable threads. It looks for the thread with the highest priority and selects that thread as the next thread to be run.

Thread priorities and policies can be listed using the following command:

```
ps -emo pid,ppid,policy,tid,comm,rtprio,cputime
```

where policy:

- `TS` is SCHED_OTHER
- `RR` is SCHED_RR
- `FF` is SCHED_FIFO
- `-` has no policy reported

The output looks like this example:

```
  PID  PPID POL   TID COMMAND         RTPRIO     TIME
18314 30285 -       - java                 - 00:01:40
    -     - RR  18314 -                    6 00:00:00
    -     - RR  18315 -                    6 00:01:40
    -     - FF  18318 -                   88 00:00:00
    -     - RR  18323 -                    6 00:00:00
    -     - FF  18324 -                   13 00:00:00
    -     - RR  18325 -                    6 00:00:00
    -     - RR  18326 -                    6 00:00:00
    -     - FF  18327 -                   11 00:00:00
    -     - FF  18328 -                   89 00:00:00
```

This output shows the Java process, the scheduling policy in force, the main thread with priority "-" (other), and some real-time threads with priorities from 11 to 89.

SCHED_OTHER
The default universal time-sharing scheduler policy that is used by most threads. These threads must be assigned with a priority of zero.

SCHED_OTHER uses time slicing, which means that each thread runs for a limited time period, after which the next thread is allowed to run.

SCHED_FIFO
Can be used only with priorities greater than zero. This usage means that when a SCHED_FIFO process becomes available it preempts any normal SCHED_OTHER thread.

If a SCHED_FIFO process that has a higher priority becomes available, it preempts an existing SCHED_FIFO process if that process has a lower priority. This thread is then kept at the top of the queue for its priority.

There is no time slicing.

SCHED_RR
Is an enhancement of SCHED_FIFO. The difference is that each thread is

allowed to run only for a limited time period. If the thread exceeds that time, it is returned to the list for its priority.

For details on these Linux scheduling policies, see the man page for **sched_setscheduler**. To query the current scheduling policy, use **sched_getscheduler**, or the ps command shown in the example.

For more information about processes, see "Examining process information" on page 128.

Related concepts

"Launching secondary processes" on page 20
The java.lang.Runtime.exec methods in the Java virtual machine (JVM) API give your Java application the ability to execute a command in a separate process.

Regular Java thread priorities and policies

Regular Java threads, that is, threads allocated as `java.lang.Thread` objects, use the default scheduling policy of SCHED_OTHER. From WebSphere Real Time for RT Linux V2 SR3, you can run regular Java threads with the SCHED_RR or SCHED_FIFO scheduling policy.

By default, Java threads are run using the default SCHED_OTHER policy. This policy maps Java threads to the operating system priority 0.

Using the SCHED_RR or SCHED_FIFO policy gives you finer control over your application, which can improve the real-time performance of Java threads. The JVM detects the priority and policy of the main thread when Java is started with the SCHED_RR or SCHED_FIFO policy. The JVM alters the priority and policy mappings accordingly. All Java threads are run at the same operating system priority as the main thread. Although SCHED_RR or SCHED_FIFO can be assigned priorities 1 - 99, the usable SCHED_RR or SCHED_FIFO priorities for WebSphere Real Time for RT Linux V2 are priorities 1 - 10. If the priority is set higher than 10, the priority of the main thread is lowered to 10 and the mapping applied based on the value of 10.

One way to alter the real-time scheduling property of a process on the command line is to use the command chrt. In the following example, the priority of the main Java thread is set to use the SCHED_FIFO scheduling policy, with an operating system priority of 6.

```
chrt -f 6 java
```

You might need to configure your system to allow priorities to be changed. See "Configuring the system to allow priority changes" on page 19 for more information.

Table 5. Java and operating system priorities

Java priority	Java priority value for thread	Operating system priority
1	MIN_PRIORITY	6
2		6
3		6
4		6
5	NORM_PRIORITY (default)	6

Table 5. Java and operating system priorities (continued)

Java priority	Java priority value for thread	Operating system priority
6		6
7		6
8		6
9		6
10	MAX_PRIORITY	6

All threads associated with the main Java thread are run at the same operating system priority. The metronome alarm thread, and other threads internal to the JVM with precise real-time timing requirements, might run at a priority higher than the priorities used by regular Java threads.

If you run the command `chrt -f 11 java`, the result is the same as running `chrt -f 10 java`. This is because you cannot apply a priority above 10 to the priority mapping used by JVM threads, although the thread that launches the JVM and waits for JVM termination remains at priority 11.

For more information about the chrt command, see http://publib.boulder.ibm.com/infocenter/lnxinfo/v3r0m0/index.jsp?topic=/liaai/realtime/liaairtchrt.htm.

Configuring the system to allow priority changes

By default, non-root users on Linux cannot raise the priority of a thread or process. You can change the system configuration to allow priority changes using the `pam_limits` module of the Pluggable Authentication Modules (PAM) for Linux.

If you cannot change the priority of a thread or process using the chrt utility, you typically see the following message:

```
sched_setscheduler: Operation not permitted
```

On recent Linux kernels, you can change the configuration of the system to allow priority changes using the `pam_limits` module. This module allows you to configure the limits on system resources, which are taken from the limits configuration file. The default file is `/etc/security/limits.conf`.

An entry in the `/etc/security/limits.conf` file has the following form:

```
<domain> <type> <item> <value>
```

where:

`<domain>` is either:

- a user name on the system that can alter limits on a resource.
- a group name, with the syntax `@group`, whose members can alter limits on a resource.
- a wildcard "`*`", which indicates that any user or group can alter limits on a resource.

`<type>` is either:

- `hard`, where hard limits are enforced by the kernel.

- `soft`, where soft limits apply, which can be moved within the range provided by the hard limits.

- a dash "`-`", which indicates hard and soft limits.

`<item>` is:

- a resource. Use `rtprio` for real-time priorities.

`<value>` is:

- a limit. Use a value in the range 1 - 100 to indicate the maximum limit for real-time priority setting.

For example,

```
* - rtprio 100
```

allows all users to change the priority of real-time processes, using chrt or other mechanisms.

By default, the root user can increase real-time priorities without limits. To apply a limit to root, the root user must be explicitly specified. Group and wildcard limits in the configuration file do not apply to the root user.

If you specify individual user limits in the file, these limits have priority over group limits.

Changes to `limits.conf` do not take effect immediately. You must restart the affected services or reboot the system for a configuration change to take effect.

For some examples of using chrt on a real-time Linux system, see http://publib.boulder.ibm.com/infocenter/lnxinfo/v3r0m0/index.jsp?topic=/liaai/realtime/liaairtchrt.htm . To enable priority changes on a real-time Linux system you can add a user to the `realtime` group, which has an entry in the `limits.conf` file.

Related concepts

“Launching secondary processes”
The java.lang.Runtime.exec methods in the Java virtual machine (JVM) API give your Java application the ability to execute a command in a separate process.

Launching secondary processes

The java.lang.Runtime.exec methods in the Java virtual machine (JVM) API give your Java application the ability to execute a command in a separate process.

From that method call, a new java.lang.Process object is created. The object can be used to control the new process, or to obtain information about it.

Several threads are created by the exec methods for this purpose. In IBM WebSphere Real Time for RT Linux, modifications of the procedure enable more deterministic behavior in a real-time environment.

The Runtime.exec call creates a “reaper” thread for each forked subprocess. The reaper thread is the only thread that waits for the subprocess to terminate. When the subprocess terminates, the reaper thread records the subprocess exit status. The reaper thread spawns the new process, and gives it the same scheduling parameters as the thread that originally called Runtime.exec.

If the spawned process is another WebSphere Real Time for RT Linux JVM, and the Runtime.exec method was called by another method running with a Linux real-time policy and priority, then the main thread of the new virtual machine maps its policy and priority to the same Linux real-time policy and priority. The priority mapping is constrained by an upper bound of 10.

The reaper thread also creates two new threads that listen to the `stdout` and `stderr` streams of the new process. These new threads are regular Java threads, not real-time Java threads. The `stdout` and `stderr` data is saved into buffers used by these threads. The buffers persist beyond the lifetime of the spawned process. This persistence allows the resources held by the spawned process to be cleared immediately when the process terminates.

If you want the `stdout` and `stderr` reader threads to run at a Linux real-time priority, launch the original JVM owning these threads with a Linux SCHED_FIFO or SCHED_RR policy and priority. The effect is to map all regular threads to a real-time policy and priority as high as 10 in the Linux operating system.

Real-time Java thread priorities and policies

Real-time threads, that is, threads allocated as `java.realtime.RealtimeThread`, and asynchronous event handlers use the SCHED_FIFO scheduling policy.

Thread scheduling and dispatching of real-time Java threads is part of the Real Time Specification for Java (RTSJ). This topic, including the scheduling policies and priority handling of real-time Java threads is discussed in the section Chapter 7, “Support for RTSJ,” on page 61.

Chapter 4. Using compiled code with WebSphere Real Time for RT Linux

IBM WebSphere Real Time for RT Linux supports several models of code compilation, providing varying levels of code performance and determinism.

Interpreted operation
This is the simplest code compilation model. The interpreter runs a Java application, but does not use code compilation at all. The interpreter exhibits good determinism, but provides very low performance, therefore avoid using this mode of operation for production systems.

To use interpreted operation, specify the **-Xint** option on the Java command line.

Low priority Just-In-Time (JIT) compilation
The default compilation model in WebSphere Real Time for RT Linux uses a Just-In-Time compiler to compile the important methods of a Java application while the application runs. In this mode, the JIT compiler works in a similar way to the operation of the JIT compiler in a non-real-time JVM. The difference is that the WebSphere Real Time for RT Linux JIT compiler runs at a lower priority level than any real-time threads. The lower priority means that the JIT compiler uses system resources when the application does not need to perform real-time tasks. The effect is that the JIT compiler does not significantly affect the performance of real-time tasks.

The JIT compiler uses two threads for compilation-related activities: the compilation thread, and the sampler thread. These threads run at lower priority than real-time tasks. The compilation thread runs in an asynchronous fashion to the application. This means that an application thread does not wait for the compilation thread to finish compiling a method at any time. The sampler thread periodically sends an asynchronous message to the application threads to identify the currently running method on each thread. Processing the message takes little time on the application thread. No messages are sent if the sampling thread cannot run because of higher priority real-time tasks. Using the JIT compiler has a small effect on determinism, but this compilation mode provides the best performance for many users.

To run an application with the JIT at low priority, see “Enabling the JIT” on page 45.

Ahead-Of-Time (AOT) precompiled code
WebSphere Real Time for RT Linux compiles Java methods to native code in a precompilation step before running the application. The precompilation step used the jxeinajar tool to compile methods using an Ahead-Of-Time compiler, and stored the results in special Java executable files. These files might be collected into bound jar files. When running an application, bound jar files are added to the application class path so that the JVM can load AOT code when the classes for methods were loaded from the JXE. Using this approach, the JIT compiler is made completely unavailable by specifying the **-Xnojit** option on the command-line. The application can use any precompiled AOT code that had been, and the interpreter for other methods. This mode of operation provides high

determinism because the JIT compiler is not present, so there are no sampling thread or context switch performance reductions. The difficulty of compiling Java code ahead of time, while conforming with the Java specification, means that AOT compiled code typically performs slower than JIT compiled code, although it is typically much faster than interpreting.

WebSphere Real Time for RT Linux V2 stores AOT code in a shared class cache rather than in JXE files, using the shared classes technology provided in the IBM Java 6 JVMs. The admincache tool lets you query the contents of a cache, list all existing caches, and populate a cache with classes and AOT code. The advantages of storing AOT compiled code are that the application jar files are not modified, and no class path changes are needed when running the application.

A shared class cache has a practical size limit, based on the available virtual address space. This means that AOT compilation for all jar files is not practical. Selective AOT compilation must be performed.

When an application runs with AOT code in a shared class cache, the AOT code for methods of a class loads automatically when the class loads into the JVM. The additional cost in loading a class to install AOT code for its methods makes it important to preload as many classes as possible before the performance critical parts of the application run.

Using AOT precompiled code provides the highest level of determinism, with good performance. AOT code can be used when your application runs by specifying the **-Xshareclasses** and **-Xaot** options. The **-Xaot** option is on by default.

To store and use AOT code with a shared class cache using the admincache tool, see "Using the admincache tool" on page 28.

For an example of running an application with AOT compiled code, see "Running the sample application using AOT" on page 80.

Mixed mode, combining AOT precompiled code and low priority JIT compilation

AOT and JIT compiled code can be used together while the application runs. This mode of operation can provide very good determinism with good performance, and very high performance for methods that run frequently. The main benefit of this mode is that AOT precompilation is used to ensure that the most important parts of your application never run in the interpreter, which is typically far slower than AOT or JIT compiled code. You do not need to precompile every method because the JIT compiler can identify dynamically any interpreted methods that run frequently, without disturbing the application performance significantly. Mixed mode is the default mode when the **-Xshareclasses** option is added to the command line.

To run your application with mixed AOT and JIT compilation, see "Running the sample application using AOT" on page 80

Managing compilation explicitly

In compilation modes with the JIT compiler enabled, the java.lang.Compiler API can be used to control JIT compiler operation explicitly. The JIT compiler compiles methods of the class passed using the compileClass() method. compileClass() is synchronous, therefore it does not return until the supplied methods have been compiled. An application might use compileClass() in an initialization phase, by iterating over the

classes used by the main phase of the application run time. When the initialization phase finishes, call the Compiler.disable() method to disable the compilation and sampling threads entirely. The main difficulty with this technique is the problem of managing the list of classes to load and compile in the application initialization phase, especially during application development.

For more information about managing compilation in an application, see IBM Real-Time Class Analysis Tool for Java.

Overview of Compilation Command-Line Options

You can run an application with JIT enabled using the **-Xjit** option or without the JIT using the **-Xnojit** option. **-Xjit** is the default mode.

You can run an application with AOT code enabled using the **-Xshareclasses -Xaot** options. Disable AOT code by using the **-Xnoaot** option. **-Xaot** is the default option, but has no effect unless the **-Xshareclasses** option is also specified, because AOT code must be stored in a shared class cache.

Limitations

You cannot use AOT precompiled code in combination with the **-Xdebug** option because use of **-Xdebug** requires code compiled in a different way. If you use **-Xdebug** in a mode where AOT is active, the AOT code is ignored and the methods are either interpreted or JIT compiled, depending on whether the JIT compiler is active.

If you want to debug your code, use interpreted or JIT-only modes.

Related reference

"Ahead-of-time options" on page 329
The definitions for the ahead-of-time options.

The ahead-of-time compiler

Ahead-of-Time (AOT) compilation allows you to compile Java classes and methods before you run your code. AOT compilation avoids the unpredictable timing effect the JIT compiler can have on sensitive performance paths. To ensure your code is compiled before it executes and for the highest level of deterministic performance, you can precompile your code into a shared class cache using the AOT compiler.

Note: AOT-compiled code does not typically run as quickly as JIT compiled code.

The Just-In-Time (JIT) compiler runs as a high-priority SCHED_OTHER thread, running above the priority of standard Java threads, but running below the priority of real-time threads. Just-in-time compilation, therefore, does not cause nondeterministic delays in real-time code. As a result, important real-time work is performed on-time because it won't be preempted by the JIT compiler. Real-time code might run as interpreted code, however, because the JIT has not had enough time to compile the hot methods that have queued up. A comparison is shown, Figure 2 on page 26.

In general, if your application has a warm-up phase, it is more efficient to run with the JIT and, if necessary, disable the JIT when the warm-up phase is complete. This approach allows the JIT compiler to generate code for the environment in which your application runs.

If the application has no warm-up phase and it is not clear if key paths of execution are compiled through standard application operation, AOT compilation works well in this environment.

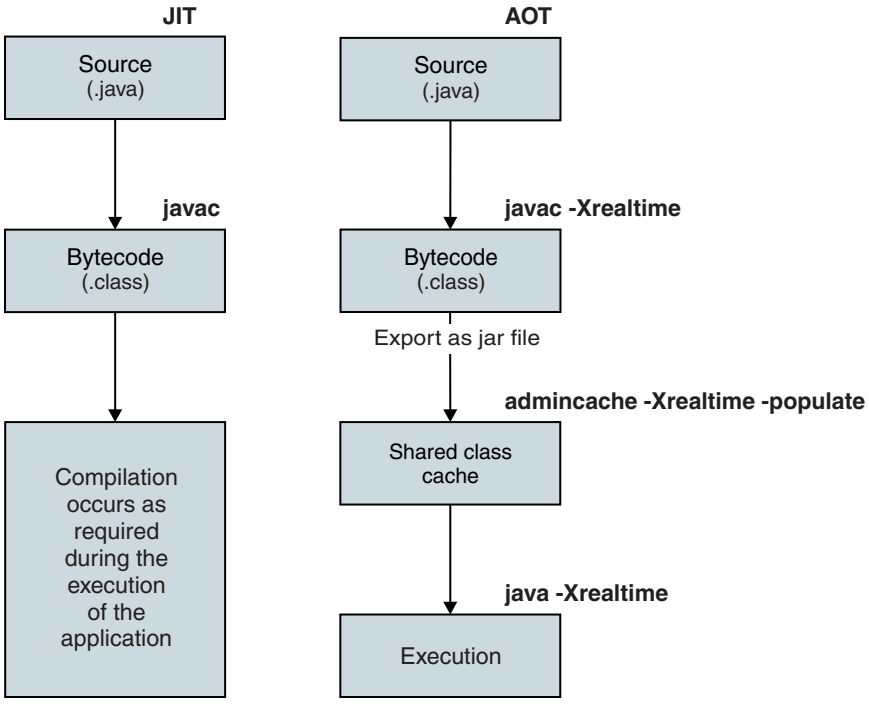


Figure 2. Comparing JIT compiler and AOT compiler.

Related tasks

"Storing precompiled jar files into a shared class cache" on page 40
You can store all, some, or include Java classes provided by IBM into a shared class cache. This process uses the **-Xrealtime** option with javac and the admincache tool to store the classes into a shared class cache.

"Precompiling all classes and methods in an application" on page 41
This procedure precompiles all the classes in an application. It stores a set of jar files into a shared class cache. All methods in all classes in those jar files are stored into the cache. The optimized jar files have all methods compiled.

"Precompiling frequently used methods" on page 42
You can use profile-directed AOT compilation to precompile only the methods that are frequently used by the application. AOT compilation stores a set of jar files into a shared class cache using an option file generated by running the application with a special option **-Xjit:verbose={precompile},vlog=optFile**. Only the methods listed in the option file are precompiled.

"Precompiling files provided by IBM" on page 43
You can precompile files provided by IBM, for example `rt.jar`, to achieve a compromise between performance and predictability.

"Using the AOT compiler"
Use these steps to precompile your Java code. This procedure describes the use of the **-Xrealtime** option in a javac command, the admincache tool, and the **-Xrealtime** and **-Xnojit** options with the java command.

"Running the sample application using AOT" on page 80
This procedure runs a standard Java application in a real-time environment using the ahead-of-time (AOT) compiler, without the need to rewrite code. Use this sample to compare running the same application using the JIT compiler.

Related reference

"Ahead-of-time options" on page 329
The definitions for the ahead-of-time options.

Using the AOT compiler

Use these steps to precompile your Java code. This procedure describes the use of the **-Xrealtime** option in a javac command, the admincache tool, and the **-Xrealtime** and **-Xnojit** options with the java command.

About this task

Using the ahead-of-time compiler means that compilation is separate from the run time of the application. Also, you can compile more methods at the same time rather than just the frequently used methods. You can compile everything in an application or just individual classes, as shown in the following steps.

Note: When using shared class caches, the name of the cache must not exceed 53 characters.

Procedure

1. From a shell prompt, enter:

   ```
   javac -Xrealtime source
   ```

   This command creates the Java bytecode from your source for use in the real-time environment. See Figure 2 on page 26.
2. Package the class files generated into a jar file. For example, to create test.jar:

```
jar cvf test.jar source
```

3. From a shell prompt, enter:

   ```
   admincache -Xrealtime -populate -aot test.jar -cacheName myCache -cp test.jar
   ```

   This command precompiles the test.jar file and writes the output to the output directory `./aot`.

4. From a shell prompt, enter: To run the file using the AOT code in the shared class cache, in a shell prompt enter:

   ```
   java -Xrealtime -Xshareclasses:name=myCache -cp test.jar -Xnojit MyTestClass
   ```

   To run the file using the AOT code in the shared class cache, recompile frequently called methods, and without creating a new jar file, n a shell prompt enter:

   ```
   java -Xrealtime -Xshareclasses:name=myCache -cp test.jar MyTestClass
   ```

   These commands use the same jar files that you precompiled in step 3.

Related concepts

"The ahead-of-time compiler" on page 25
Ahead-of-Time (AOT) compilation allows you to compile Java classes and methods before you run your code. AOT compilation avoids the unpredictable timing effect the JIT compiler can have on sensitive performance paths. To ensure your code is compiled before it executes and for the highest level of deterministic performance, you can precompile your code into a shared class cache using the AOT compiler.

Related tasks

"Running the sample application using AOT" on page 80
This procedure runs a standard Java application in a real-time environment using the ahead-of-time (AOT) compiler, without the need to rewrite code. Use this sample to compare running the same application using the JIT compiler.

Related reference

"Ahead-of-time options" on page 329
The definitions for the ahead-of-time options.

Using the admincache tool

The admincache tool is used to manage the shared class caches on a workstation.

In the IBM WebSphere Real Time for RT Linux product, the admincache tool can be used to create a shared class cache containing classes or classes and AOT compiled code. After caches have been created, this tool can also be used to inspect existing caches.

The shared class cache is used to reduce the memory footprint in multi-JVM scenarios, and to accelerate application start-up.

Shared class caches can be used by WebSphere Real Time for RT Linux in both non-real-time and real-time modes, but the cache format, creation and population techniques differ. Real-time mode caches are not compatible with non-real-time mode caches. In non-real-time mode, caches are created and populated in the same way as the standard JVM. This means the cache is created and populated by the JVM as it runs an application, transparently to the user. In real-time mode, using the **-Xrealtime** option, shared class caches must be created and prepopulated by admincache, using the **-populate** option. Applications running in real-time mode might read content from the prepopulated cache, but cannot modify its contents.

Shared class caches created in the real-time mode can only be used when running an application in real-time mode. Shared class caches created in the non-real-time mode can only be used when running an application in non-real-time mode. This also applies to the admincache tool. To manage caches created by the JVM in real-time mode, use admincache with the **-Xrealtime** option. To manage caches created by the JVM in non-real-time mode, do not use the **-Xrealtime** option. To connect to a shared class cache at run time, add the **-Xshareclasses** option to the command-line.

Multiple shared class caches can be created on a workstation, each with a specific name and in a specific directory. When a new cache is created, the name of the cache can be specified by the **-cacheName** <*name*> option. The cache name must not exceed 53 characters.

By default, shared class caches are created in the `/tmp/javasharedresources` directory, but this location can be overridden by specifying the **-cacheDir** <*directory*> option. The internal format for a shared class caches depends on the characteristics of the workstation where it is created. This means shared class caches cannot be created on networked drives as a safety measure. Another reason for this restriction is the slower and unpredictable performance effects when accessing a shared class cache from a networked file system.

If no cache name is specified on the command line, the default is **sharedcc_<user_login>**

For more information about shared cache operation in the non-real-time mode, see "Class data sharing between JVMs for non Real-Time" on page 7.

Note: From IBM WebSphere Real Time for RT Linux V2 SR1 and above, you must use the **-classpath** option with the **-populate** option.

Creating a Real-Time Shared Class Cache

The admincache tool is used to create shared class caches that are accessible in real-time mode.

Note: You must be aware of the security considerations when creating shared class cache files with default settings. See "Security considerations for the shared class cache" on page 7 for more information about security considerations for the shared class cache and information about changing the default permissions.

The admincache tool **-populate** option is used to create shared class caches. The option is used in combination with a list of jar files, or a directory, or a directory tree to search for jar files. For each jar file specified or found, admincache stores each class in the jar file in the shared class cache. The class methods are also AOT compiled and stored in the shared class cache, unless you specify the **-noaot** option.

You must use the **-classpath** option with **-populate** or you will see the following error message:

```
-populate action requires -classpath <class path> option to be specified
```

The admincache **-help** option lists the suboptions that can be used to control how admincache populates the cache.

```
$ admincache -Xrealtime -help
Usage: admincache [option]*
  where [option] can be:
     -help | -?                Action: show this help
     -[no]logo                 show copyright message
     -Xrealtime                use in real time environment
     -cacheName <name>         specify name of shared cache (Use %%u to substitute username)
     -cacheDir <dir>           set the location of the JVM cache files
     -listAllCaches            Action: list all existing shared class caches
     -printStats               Action: print cache statistics
     -printAllStats            Action: print more verbose cache statistics
     -destroy                  Action: destroy the named (or default) cache
     -destroyAll               Action: destroy all caches
     -populate                 Action: Create a new cache and populate it
              -searchPath <path>      specify the directory in which to find files if no files speci
                                        only one -searchPath option can be specified
             -classpath <class path> specify the classpath that will be used at runtime to access th
                                        the -classpath option is required
       -[no]recurse            [do not] recurse into subdirectories to find files to convert
       -[no]grow               if specified cache exists already, [do not] add to it
                                        if nogrow is selected, the existing cache will be reset
       -verbose                print out progress messages for each jar
       -noisy                  print out progress messages for each class in each jar
       -quiet                  suppress all progress messages
       -[no]aot                also perform AOT compilation on methods after storing classes into ca
       -aotFilter <signature>  only matching methods will be AOT compiled and stored into cache
       -aotFilterFile <file>   only methods matching those in file will be AOT compiled and stored i
                                        (input file must have been created by -Xjit:verbose={precompile},vl
       -printvmargs            print VM arguments needed to access populated cache at runtime
       [jar file]*.[jar][zip]  explicit list of jar files to populate into cache
                                        if no files are specified, all files.[jar][zip] in the searchPath w
Exactly one action option must be specified
```

Note: When using shared class caches, the name specified by the **-cacheName** option must not exceed 53 characters.

A list of jar files can be specified, in which case only the classes from those jar files will be added to the shared class cache. If you do not specify a list of jar files, use the **-searchPath <path>** option to specify a directory tree to search for .jar or .zip files. The **-recurse** option is the default, and means that the directory tree is searched recursively for .jar or .zip files. The **-norecurse** options means that only the specified directory is searched. Specify the **-classpath <class path>** option so that admincache can find all the classes needed to process the specified jar files. The classes are loaded into the JVM as part of populating the shared class cache, so it is important that all referenced classes and superclasses can be found by admincache when it tries to load a class from a jar file.

The **-grow** option specifies that a new jar file is added to the existing cache contents, if there is an existing shared class cache of the same name in the cache directory. The **-nogrow** option specifies that a new jar file replaces the old cache contents, if there is an existing shared class cache of the same name in the old cache directory. The **-grow** option is used to add new jar files that do not currently exist in the shared class cache, not to replace classes that have changed. Do not use the **-grow** option to update classes that are already in the cache but have changed because of application modifications. To update existing classes, create a completely new cache with the current class contents. If you do not update your shared class cache when you change a class, your application will run properly with the new class contents, but will not be taking advantage of the shared class cache. This is because the changed class will be loaded from disk rather than from

the shared class cache. Loading the class from disk means that AOT compiled code cannot be used for that class. Regenerate your shared class cache when you change a class.

Use the **-quiet**, **-verbose**, and **-noisy** options to control the level of detail provided by admincache.

To specify Ahead-Of-Time (AOT) precompilation for the methods in the classes populating the shared class cache, use the **-aot** option. To prevent AOT precompilation and only store classes into the shared class cache, use the **-noaot** option. The **-aot** option is the default setting.

To precompile some methods selectively, use the **-aotFilter** <*signature*> or **-aotFilterFile** <*file*> options. The <*signature*> is a simplified regular expression for a method signature, enclosed in curly braces, where '`*`' can replace any sequence of characters. You might need to enclose <*signature*> in single quotation marks so that the shell does not interpret any of the characters in the method signature.

Table 6shows some examples of the <*signature*> option.

Table 6. Examples of the <signature> option

Signature	Meaning
`-aotFilter '{java/lang/*}'`	AOT compiles methods in the java/lang package.
`-aotFilter '{*.sample*}'`	AOT compiles methods beginning with "sample".
`-aotFilter '{mypackage/myclass.mymethod(I)I}'`	AOT compiles the method with this exact signature.

The **-aotFilterFile** <*file*> option uses the contents of <*file*> to select the methods for AOT compilation. No other methods are AOT compiled. The contents of <*file*> are generated during an earlier run of the application using the **-Xjit:verbose={precompile},vlog=**<*file*> option. The verbose output stored in <*file*> uses an internal format. This format is required by the **-aotFilterFile** option.

Note: The **-vlog=**<*file*> option does not directly generate a file called "file". A date and process ID string is appended to "file" when the verbose output is generated. By specifying the option **-Xjit:verbose={precompile},vlog=my_file**, the generated file name is similar to `my_file.<date>.<#>.<process id>` The extra fields make it easier to generate individual verbose log files in multiple-JVM scenarios where it can be difficult to supply command-line options to one particular JVM or to use different **-Xjit** command-line options with different JVMs. In a single JVM scenario, these numbers are appended to the file name supplied on the command-line.

A generated file can be used with the **-aotFilterFile** option, without requiring any editing. Multiple verbose log files generated by several application runs using the **-Xjit:verbose={precompile},vlog=<file>** option can be concatenated and supplied to admincache using the **-aotFilterFile** option.

The **-printvmargs** option helps ensure that the correct arguments are supplied on the command line when the application runs.

```
$ admincache -Xrealtime -classpath myapp.jar -cacheDir myCacheDir -cacheName myCache -populate mya
J9 Java(TM) admincache 1.0
Licensed Materials - Property of IBM
```

```
(c) Copyright IBM Corp. 1991, 2008  All Rights Reserved
IBM is a registered trademark of IBM Corp.
Java and all Java-based marks and logos are trademarks or registered
trademarks of Sun Microsystems, Inc.

JVMSHRC023E Cache does not exist
Converting files
Converting /home/user/myapp.jar
Succeeded to convert jar file /home/user/myapp.jar

Processing complete

VM args needed at runtime: -Xshareclasses:name=myCache,cacheDir=myCacheDir -classpath myapp.jar -Xao
```

In this example, the final line of output shows the options that should be added to the command line when running the application, so that the classes and AOT methods stored in the shared class cache are used. To use the options from this example, enter the command:

```
java -Xshareclasses:name=myCache,cacheDir=myCacheDir -classpath myapp.jar -Xaot myMainClass <applica
```

Managing Shared Class Caches with admincache

The admincache tool includes several utilities to manage the shared class caches on your system.

The admincache tool provides utilities to help with several activities.

- Listing the shared class caches present in a cache.
- Providing details about the contents of a shared class cache.
- Removing some or all of the caches in a specific cache directory.

Listing Available Shared Class Caches:

The admincache tool provides a list of shared class caches present in a cache.

To obtain a list of all shared class caches present in a cache, use the **-listAllCaches** option and specify the cache directory using the **-cacheDir** option.

```
$ admincache -Xrealtime -listAllCaches
J9 Java(TM) admincache 1.0
Licensed Materials - Property of IBM

(c) Copyright IBM Corp. 1991, 2008  All Rights Reserved
IBM is a registered trademark of IBM Corp.
Java and all Java-based marks and logos are trademarks or registered
trademarks of Sun Microsystems, Inc.

Listing all caches in cacheDir /tmp/javasharedresources/

Cache name              level         persistent  last detach time

Compatible shared caches
sharedcc_username       Java6 32-bit  yes         Thu Oct 16 17:02:39 2008
rtCache                 Java6 32-bit  yes         Thu Oct 16 17:03:12 2008

Incompatible shared caches
nonrtCache              Java6 32-bit  yes         Thu Oct 16 17:17:32 2008
```

In this example, there are two compatible shared class caches in the default cache directory:

- The default named cache for a user with the login *username*
- Another cache called *rtCache*

The example also shows an incompatible cache called *nonrtCache*. The *nonrtCache* was created by the JVM while executing in the non-real-time mode. This means it cannot be accessed using the **-Xrealtime** option.

The real-time mode JVM can see caches created in non-real-time mode. The non-real-time mode JVM cannot see caches created in real-time mode.

```
$ admincache -listAllCaches
J9 Java(TM) admincache 1.0
Licensed Materials - Property of IBM

(c) Copyright IBM Corp. 1991, 2008  All Rights Reserved
IBM is a registered trademark of IBM Corp.
Java and all Java-based marks and logos are trademarks or registered
trademarks of Sun Microsystems, Inc.


Listing all caches in cacheDir /tmp/javasharedresources/

Cache name              level         persistent  last detach time

Compatible shared caches
nonrtCache              Java6 32-bit  yes         Thu Oct 16 17:17:32 2008
```

In this example, *nonrtCache* is listed, and it is shown as compatible because **-Xrealtime** is not specified.

Inspecting Contents of Shared Class Caches:

The admincache tool provides a description of a shared class cache contents.

You can use the admincache tool **-printStats** option to obtain an overview describing the main contents of a shared class cache. For information about a specific cache, in a specific cache directory, use the **-cacheName** and **-cacheDir** options. The following example gives information about the *nonrtCache* cache in the default cache directory.

```
$ admincache -cacheName nonrtCache -printStats
J9 Java(TM) admincache 1.0
Licensed Materials - Property of IBM

(c) Copyright IBM Corp. 1991, 2008  All Rights Reserved
IBM is a registered trademark of IBM Corp.
Java and all Java-based marks and logos are trademarks or registered
trademarks of Sun Microsystems, Inc.


Current statistics for cache "nonrtCache":


base address       = 0xD5445000
end address        = 0xD6437000
allocation pointer = 0xD5529FA8

cache size         = 16776852
free bytes         = 14070360
ROMClass bytes     = 1166004
AOT bytes          = 1437412
Data bytes         = 57440
Metadata bytes     = 45636
Metadata % used    = 1%

# ROMClasses       = 372
# AOT Methods      = 981
# Classpaths       = 1
```

```
# URLs              = 0
# Tokens            = 0
# Stale classes     = 0
% Stale classes     = 0%

Cache is 16% full
```

Note: When using shared class caches, the name of the cache must not exceed 53 characters.

There are several pieces of useful information about this cache:

- The size of the cache, shown as `cache size = 16776852`.
- The space is available in the cache, shown as `free bytes = 14070360`. This means the cache is approximately 16% full.
- The number of classes stored in the cache, shown as `# ROMClasses = 372`.
- The number of AOT methods stored in the cache, shown as `# AOT Methods = 981`.

For more details about the information provided by the **-printStats** option in the admincache tool, see "printStats utility" on page 269.

The **-printAllStats** option provides a more detailed description of the contents of a shared class cache. The information includes the list of classes and AOT methods store in the cache. Output from the **-printAllStats** option is verbose.

Classes contained in the cache are indicated by lines similar to:

```
1: 0xD643B788 ROMCLASS: java/lang/ClassLoader at 0xD5469B88.
```

This line indicates that the class java/lang/ClassLoader is contained in the cache. The addresses are internal to the shared class cache, and are rarely useful except for diagnostic purposes.

AOT methods contained in the cache are indicated by lines similar to:

```
1: 0xD643B290 AOT: callerClassLoader
        for ROMClass java/lang/ClassLoader at 0xD5469B88.
```

These lines indicate that the callerClassLoader method from the java/lang/ClassLoader class is contained in the class. The addresses listed are internal shared cache addresses. Output from the **-printAllStats** option does not show the signature for each AOT method in the cache, where the signature consists of the parameter types and return type.

For more details about the information provided by the **-printAllStats** option in the admincache tool, see "printAllStats utility" on page 270.

Related information

"printAllStats utility" on page 270
The printAllStats utility is a suboption of **-Xshareclasses**, optionally taking a cache name using **name=**<*name*>. This utility lists the cache contents in order. It aims to give as much diagnostic information as possible and, because the output is listed in chronological order, you can interpret it as an "audit trail" of cache updates. Because it is a cache utility, the JVM displays the information on the cache specified or the default cache and then exits.

"printStats utility" on page 269
The printStats utility is a suboption of **-Xshareclasses**, optionally taking a cache name using **name=**<*name*>. It prints summary information on the cache specified to standard error. Because it is a cache utility, the JVM displays the information on the cache specified and then exits.

"Security considerations for the shared class cache" on page 7
The shared class cache is designed for ease of cache management and usability, but the default security policy might not be appropriate.

Destroying Shared Class Caches:

The admincache tool has options to erase a particular cache or all caches in a specified cache directory.

The admincache tool **-destroy** option is used to erase a particular cache in a specific cache directory, if the user has permission to do so. The **-destroyAll** option is used to erase all the caches, if the user has permission to do so. For example:

```
$ admincache -Xrealtime -destroy
J9 Java(TM) admincache 1.0
Licensed Materials - Property of IBM

(c) Copyright IBM Corp. 1991, 2008  All Rights Reserved
IBM is a registered trademark of IBM Corp.
Java and all Java-based marks and logos are trademarks or registered
trademarks of Sun Microsystems, Inc.

JVMSHRC256I Persistent shared cache "sharedcc_username" has been destroyed
```

After erasing the cache, a listing of the available shared class caches in the default cache directory shows that the erased cache no longer appears:

```
$ admincache -Xrealtime -listAllCaches
J9 Java(TM) admincache 1.0
Licensed Materials - Property of IBM

(c) Copyright IBM Corp. 1991, 2008  All Rights Reserved
IBM is a registered trademark of IBM Corp.
Java and all Java-based marks and logos are trademarks or registered
trademarks of Sun Microsystems, Inc.

Listing all caches in cacheDir /tmp/javasharedresources/

Cache name              level         persistent  last detach time

Compatible shared caches
rtCache                 Java6 32-bit  yes         Thu Oct 16 17:03:12 2008

Incompatible shared caches
nonrtCache              Java6 32-bit  yes         Thu Oct 16 17:17:32 2008
```

The **-destroyAll** option removes all caches in the specified cache directory, regardless of whether they are compatible or not with the current JVM. The **-destroyAll** option must be used with great care:

```
$ admincache -Xrealtime -destroyAll
J9 Java(TM) admincache 1.0
Licensed Materials - Property of IBM

(c) Copyright IBM Corp. 1991, 2008  All Rights Reserved
IBM is a registered trademark of IBM Corp.
Java and all Java-based marks and logos are trademarks or registered
trademarks of Sun Microsystems, Inc.

Attempting to destroy all caches in cacheDir /tmp/javasharedresources/

JVMSHRC256I Persistent shared cache "rtCache" has been destroyed
JVMSHRC256I Persistent shared cache "nonrtCache" has been destroyed
```

The result is that there are no longer any shared class caches available on the machine:

```
$ admincache -Xrealtime -listAllCaches
J9 Java(TM) admincache 1.0
Licensed Materials - Property of IBM

(c) Copyright IBM Corp. 1991, 2008  All Rights Reserved
IBM is a registered trademark of IBM Corp.
Java and all Java-based marks and logos are trademarks or registered
trademarks of Sun Microsystems, Inc.

JVMSHRC005I No shared class caches available
```

If the current user does not have permission to access a cache, then the cache is not be destroyed by either the **-destroy** or **-destroyAll** options.

Practical Sizes for Shared Class Caches

The admincache tool provides information for sizing shared class caches.

For smaller applications, a shared class cache can be populated with all classes and methods without producing a prohibitively large cache size. For larger applications, the size of the resulting shared class cache might become too large for practical purposes. This is because a JVM process must have sufficient virtual address space to address the entire contents of the shared class cache. There are some considerations you can apply when using the shared class cache technology.

The shared class cache must be virtually addressable in its entirety, in any JVM connecting to it. This means avoiding using shared class caches larger than 700 MB. The admincache tool can predict the size of a cache. If the tool indicates that the cache will be larger than the 700 MB limit, a message is displayed advising that you store a smaller number of classes, or are more selective about the AOT methods stored in the cache.

```
$ admincache -Xrealtime -populate veryBigJar.jar -cp <my class path>
J9 Java(TM) admincache 1.0
Licensed Materials - Property of IBM

(c) Copyright IBM Corp. 1991, 2008  All Rights Reserved
IBM is a registered trademark of IBM Corp.
Java and all Java-based marks and logos are trademarks or registered
trademarks of Sun Microsystems, Inc.

WARNING: predicted cache size (15960MB) exceeds recommended maximum shared class cache size of 700MB
If your jar files contain primarily class files then you may not be able to create a cache of this s
        or you may not be able to connect to the created cache when you run your application.
```

```
Alternatively, you may want to more selectively compile AOT methods by using -aotFilterFile
To override this warning message, please directly specify -Xscmx15960M on your command-line
        but beware that the resulting failure may not occur until the very end of the population p
```

The admincache tool predicts a conservative cache size, based upon the total size of the jar files specified or found for population. This means that the prediction might not be accurate if the jar file contains many files that are not class files. To get a more accurate prediction of the cache size, create temporary versions of the jar files that contain only the class files. If the admincache tool still produces a warning message, you might consider AOT precompiling the methods in the jar file more selectively, by using the **-aotFilter** <*pattern*> or **-aotFilterFile** <*file*> options. The admincache tool message reminds you that the prediction does not take into account AOT methods filtered out by these options.

To override the warning message and proceed to the cache population step, add the indicated **-Xscmx** option to the admincache command line. If the predicted size is very large, the admincache tool might not be able to create a shared class cache of the required size. To resolve this, reduce the cache size until admincache tool is able to proceed.

When the final cache is written to disk, it is only as big as is needed to hold the classes and AOT methods specified. This means that specifying a large initial cache size is not a problem.

Storing SDK classes in a Shared Class Cache

Creating a cache containing all the jar files from the SDK might not be necessary for all applications.

The number and size of the jar files in the SDK means that trying to create a cache containing all of these jar files results in a warning message advising that the resulting cache will be too large. For many applications, most of the SDK jar files are never be referenced.

The main SDK jar files are located in the `SDK/jre/lib` directory. For most applications, the most important of these jar files is `rt.jar`, which is new in Java 6 releases. `rt.jar` is a collection of classes previously stored in separate jar files before the Java 6 release. Populating a shared class cache with `rt.jar` alone, and compiling all of its methods with the AOT compiler, creates a cache approximately 300 MB in size. Most of the methods from the `rt.jar` classes will not be referenced by a typical application. To populate a shared class cache with `rt.jar`:

1. Populate the classes only from `rt.jar` into the shared class cache. This consumes approximately 50 MB of space in the cache.
2. Use the **-aotFilterFile** <*file*> option to compile only the methods your program might use. You can generate the <*file*> by running the application.

There are other commonly-used and important jar files in the SDK, including:

- `sdk/jre/lib/i386/realtime/jclSC160/realtime.jar`
- `sdk/jre/lib/i386/realtime/jclSC160/vm.jar`
- `sdk/jre/lib/java.util.jar`

`realtime.jar` contains the IBM implementation of the Real Time Specification for Java (RTSJ). If your application uses any of the features of the RTSJ, store the `realtime.jar` file in the shared class cache for better deterministic performance. `vm.jar` contains several internal JVM classes, commonly used in all applications. `java.util.jar` contains several container classes, and must be stored into every application's shared class cache for better deterministic performance.

Other jar files in sdk/jre/lib and sdk/jre/lib/ext directories can be stored into a shared class cache if an application uses those classes. The easiest way to identify if your application uses these classes is to use the **-verbose:dynload** option when running your program. The **-verbose:dynload** option describes only the classes loaded by the current run of the application. For example:

```
<Loaded java/io/InputStreamReader from /myjdk/sdk/jre/lib/rt.jar>
<  Class size 2126; ROM size 2280; debug size 0>
<  Read time 54 usec; Load time 47 usec; Translate time 86 usec>
<Loaded java/util/LinkedHashSet from /myjdk/sdk/jre/lib/java.util.jar>
<  Class size 1218; ROM size 1136; debug size 0>
<  Read time 48 usec; Load time 31 usec; Translate time 55 usec>
<Loaded java/util/HashSet from /myjdk/sdk/jre/lib/java.util.jar>
<  Class size 3171; ROM size 2664; debug size 0>
<  Read time 71 usec; Load time 70 usec; Translate time 118 usec>
```

This example output shows three classes loaded from two different SDK jar files. The java/io/InputStreamReader class was loaded from rt.jar. The java/util/LinkedHashSet and java/util/HashSet classes were loaded from java.util.jar.

WebSphere Real Time V1 Migration

How to migrate from using the jxeinajar tool to using the admincache tool.

About this task

Note: jxeinajar is not supported in WebSphere Real Time for RT Linux V2 SR1 or above. This is because it is a requirement that the admincache command must use the **-classpath** option with **-populate**; which is unavailable for jxeinajar.

The jxeinajar tool supplied with WebSphere Real Time V1 still exists in IBM WebSphere Real Time for RT Linux in versions earlier than IBM WebSphere Real Time for RT Linux SR1, but does not have the same behavior. jxeinajar now populates a shared class cache with a default name, taken from the user's login name. jxeinajar accepts the same command lines used for WebSphere Real Time V1, but is no longer the way to create AOT code. Using the jxeinajar tool might create too large a shared class cache, because the combined size of several JXEs can easily go beyond the practical size of a shared class cache. By default, the IBM WebSphere Real Time for RT Linux V2 jxeinajar tool also creates an output directory with jar files matching the original jars files. The classes and AOT methods are stored into the shared class cache, meaning that the jar files in the output path are copies of the original files.

The jxeinajar tool provided in IBM WebSphere Real Time for RT Linux V2 effectively runs the admincache tool with the default options **-populate -aot -grow**. jxeinajar tool must be run with the **-Xrealtime** option. Using the **-grow** option means that each jar file specified is added to the existing default shared class cache for the user running the tool. Before running any jxeinajar commands, run the command:

```
admincache -Xrealtime -destroy
```

This ensures that the cache starts from an empty state.

IBM WebSphere Real Time for RT Linux V2 uses a Java 6 based JVM, whereas WebSphere Real Time V1 used a Java 5 based JVM. This means that some of the SDK jar files have different names and are stored in different locations. Table 7 on page 39 provides a simple summary of where classes can be found.

Table 7. Simple summary showing new class locations

WebSphere Real Time V1 class location	IBM WebSphere Real Time for RT Linux V2 class location
sdk/jre/lib/core.jar	sdk/jre/lib/rt.jar
sdk/jre/bin/realtime/jclSC150/realtime.jar	sdk/jre/lib/i386/realtime/jclSC160/realtime.jar
sdk/jre/bin/realtime/jclSC150/vm.jar	sdk/jre/lib/i386/realtime/jclSC160/vm.jar

Some new container class implementations for Java 6 can now be found in `sdk/jre/lib/java.util.jar`.

Procedure

1. Choose an appropriate name for the cache. The name must not exceed 53 characters.
2. Modify each jxeinajar command as follows:
   a. Change the command from jxeinajar to admincache
   b. Remove the **-outPath** option and the directory it specified. This is because admincache no longer creates an output directory.
   c. Add the **-populate** option. From WebSphere Real Time for RT Linux V2 SR1 or above, you must use **-classpath** with the **-populate** option.
   d. Add the **-cacheName** <*name*> option, specifying the cache name chosen earlier. The name specified by the **-cacheName** option must not exceed 53 characters.
   e. Optional: If you want the shared class cache stored in a particular location, add the **-cacheDir** <*dir*> option.
   f. Add the **-classpath** <*paths*> option. This means that all your classes can be loaded by admincache.
   g. Add the **-printvmargs** option. This ensures that admincache reports the options needed at run time on the application command-line.
3. Remove any JXEs previously added the class path originally used with jxeinajar. When you run your application, supply the application class path in the same way as when you do not use JXEs.
4. Add or update the command-line options reported after using the **-printvmargs** option. For example:

   ```
   -Xshareclasses:name="my cache",cacheDir="my cache directory" -classpath <path to jars>
   ```

Other admincache considerations

Useful information for working with admincache.

Cache Population and Immortal Memory Sizing

When admincache populates a shared class cache in real-time mode, it must load each class as it goes. Each class consumes some immortal memory, thus it is possible that the default immortal memory size will not be large enough for all the classes requested. If the admincache tool throws an OutOfMemory error while populating a cache with many classes, try increasing the immortal memory size beyond the default 16 MB, using the **-Xgc:immortalMemorySize=32M** option.

When you Change Classes

If a class file is changed on disk, the shared class cache technology automatically detects that the cached version of that class in a shared class cache must not be used. Your program will operate correctly, but cannot take full advantage of the shared class cache, and any AOT methods for that class will not be used. If you change a class in your application, re-create your shared class cache. Do not try to use the **-grow** option to repopulate only the jar file containing the modified class, because this option is not designed for the scenario where the jar file already exists in the cache.

Managing shared caches and JXEs

Shared caches require address space even if there are no files loaded. With JXEs, it is possible to reference a JXE on your class path, but unless you load a class from that JXE it does not consume any address space or consume memory in the JVM process. See "How the IBM JVM manages memory" on page 114 for more information on how shared class caches consume memory in the JVM process. If you created a large number of JXE files with WebSphere Real Time V1, you cannot create a cache capable of holding all of those JXEs. Instead, perform a more selective population of shared class caches than JXEs, using the **-aotFilter** and **-aotFilterFile** options as well as only storing jar files into the cache if they are used by your program.

Storing precompiled jar files into a shared class cache

You can store all, some, or include Java classes provided by IBM into a shared class cache. This process uses the **-Xrealtime** option with javac and the admincache tool to store the classes into a shared class cache.

Before you begin

Jar files stored ahead of time into a shared class cache is supported only with the **-Xrealtime** option and when running java with the **-Xrealtime** option. You can use the same jar files when running with or without **-Xrealtime**, but the jar files stored into the cache can only be used when **-Xrealtime** is specified.

Note: When using shared class caches, the cache name must not exceed 53 characters.

About this task

You can store jar files into a shared class cache using the admincache tool. admincache enables you to build your application in one of three ways.

Note:
- If you have set a timeout on your Linux system, you might need to override it when precompiling large jar files; otherwise, compilation will time out and the jar file is not created.

Related concepts

"The ahead-of-time compiler" on page 25
Ahead-of-Time (AOT) compilation allows you to compile Java classes and methods before you run your code. AOT compilation avoids the unpredictable timing effect the JIT compiler can have on sensitive performance paths. To ensure your code is compiled before it executes and for the highest level of deterministic performance, you can precompile your code into a shared class cache using the AOT compiler.

Related reference

"Ahead-of-time options" on page 329
The definitions for the ahead-of-time options.

Precompiling all classes and methods in an application

This procedure precompiles all the classes in an application. It stores a set of jar files into a shared class cache. All methods in all classes in those jar files are stored into the cache. The optimized jar files have all methods compiled.

About this task

For the purposes of this example, the application resides under the directory specified by the environment variable *$APP_HOME* and the jar files are in the subdirectory *$APP_HOME*/lib. The application also uses some classes from those provided by IBM in `core.jarrt.jar`. In this case, you can precompile only the application code, namely `main.jar` and `util.jar`.

By default, the shared class cache is in `/tmp/javasharedresources`. Use the **-cacheDir** option to put the cache into a different directory. You cannot create a cache on a networked file system.

Procedure

1. From a shell prompt, enter: `cd` *$APP_HOME*

   where *$APP_HOME* is the directory of your application.
2. From a shell prompt, enter: `cd` *$APP_HOME*`/lib`. *$APP_HOME*`/lib` is the directory where `main.jar` and `util.jar` are stored.
3. From a shell prompt, enter: `admincache -Xrealtime -populate -aot -classpath` *$APP_HOME*`/lib -searchPath` *$APP_HOME*`/lib -norecurse .` This procedure optimizes each of the jar files found in *$APP_HOME*`/lib`, writing out progress information to the screen, and then creating the new jar file in the *$APP_HOME*`/aot` directory. You can specify a cache name with **-cacheName <name>**, but a default name based on the user's login is used if none is specified.

   Note: The name specified by the **-cacheName** option must not exceed 53 characters.
4. From a shell prompt, entering: `admincache -Xrealtime -listAllCaches` shows the existence of the cache.

What to do next

For more options, specify: `admincache -Xrealtime -help`.

Related concepts

"The ahead-of-time compiler" on page 25
Ahead-of-Time (AOT) compilation allows you to compile Java classes and methods before you run your code. AOT compilation avoids the unpredictable timing effect the JIT compiler can have on sensitive performance paths. To ensure your code is compiled before it executes and for the highest level of deterministic performance, you can precompile your code into a shared class cache using the AOT compiler.

Related reference

"Ahead-of-time options" on page 329
The definitions for the ahead-of-time options.

Precompiling frequently used methods

You can use profile-directed AOT compilation to precompile only the methods that are frequently used by the application. AOT compilation stores a set of jar files into a shared class cache using an option file generated by running the application with a special option **-Xjit:verbose={precompile},vlog=optFile**. Only the methods listed in the option file are precompiled.

Before you begin

Before you start, create a list of those methods that are typically compiled by a JIT compiler.

About this task

You can edit the file generated by the **-Xjit:verbose={precompile}** option. The file is an explicit specification of the methods that are to be precompiled. These methods are specific; that is, they contain the full signature for each method to be compiled, which lets you compile `com/acme/sample.myMethod(J)V` but not `com/acme/sample.myMethod(I)V`.

Note: When using shared class caches, the name of the cache must not exceed 53 characters.

Procedure

1. From a shell prompt, enter:

   ```
   cd $APP_HOME
   ```

   where *$APP_HOME* is the directory of your application.
2. From a shell prompt, enter:

```
java -Xjit:verbose={precompile},vlog=$APP_HOME/app.precompileOpts \
     -cp $APP_HOME/lib/demo.jar applicationName
```

   where:

   - *app.precompileOpts* is the name of the log file that lists the methods compiled with JIT.
   - *applicationName* is the name of your application.

   This command creates a list of the methods that are compiled using JIT.
3. From a shell prompt, enter:

   ```
   cd $APP_HOME/lib
   ```

   $APP_HOME/lib is the directory where the jar files for your application are stored.
4. To compile all the sample application methods into the cache, enter:

```
admincache -Xrealtime -populate -cacheName myCache \
           -aotFilterFile $APP_HOME/app.precompileOpts \
           -cp $APP_HOME/lib/demo.jar
```

5. To compile `realtime.jar` and `vm.jar` into the cache, enter:

```
admincache -Xrealtime -populate -grow -cacheName myCache \
           -aotFilterFile $APP_HOME/app.precompileOpts \
           -searchPath $JAVA_HOME/jre/bin/realtime/jclSC160
           -cp $APP_HOME/lib/demo.jar
```

6. To compile `rt.jar` into the cache, enter:

```
admincache -Xrealtime -populate -grow -cacheName myCache \
           -aotFilterFile $APP_HOME/app.precompileOpts \
           $JAVA_HOME/jre/lib/rt.jar
           -cp $APP_HOME/lib/demo.jar
```

7. To test this command run your application with the **-nojit** option, which uses the code in the cache. From the shell prompt, enter:

```
java -Xrealtime -Xshareclasses:name=myCache -Xnojit \
     -cp $APPHOME/aot/demo.jar applicationName
```

where *applicationName* is the name of your application.

Related concepts

"The ahead-of-time compiler" on page 25
Ahead-of-Time (AOT) compilation allows you to compile Java classes and methods before you run your code. AOT compilation avoids the unpredictable timing effect the JIT compiler can have on sensitive performance paths. To ensure your code is compiled before it executes and for the highest level of deterministic performance, you can precompile your code into a shared class cache using the AOT compiler.

Related reference

"Ahead-of-time options" on page 329
The definitions for the ahead-of-time options.

Precompiling files provided by IBM

You can precompile files provided by IBM, for example `rt.jar`, to achieve a compromise between performance and predictability.

About this task

The precompilation is similar to the task of precompiling your application jars but an additional requirement applies at run time; you must ensure that your boot class path is specified correctly to use these files instead of the files in the JRE. You can do this with the **-Xshareclasses** option, which instructs the JVM to look first in the specified class cache ahead of the default class path locations.

Note: When using shared class caches, the name of the cache must not exceed 53 characters.

Precompile `rt.jar` for use with the application:

Procedure

1. From a shell prompt, enter: `cd` *$JAVA_HOME*`/lib` where *$JAVA_HOME* is your Java home directory.
2. Run the admincache tool. At a shell prompt, enter:

   ```
   admincache -Xrealtime -populate -cacheName myCache -classpath <class path> rt.jar
   ```

   This command populates the cache called `myCache` with the results of precompiling the IBM-provided file called `rt.jar`.

3. Run your application specifying the **-Xshareclasses** option to specify the cache name. To run your application, enter:

```
java -Xrealtime -Xnojit -Xshareclasses:name=myCache
        -classpath:$APP_HOME/main.jar:$APP_HOME/util.jar ...
```

Related concepts

"The ahead-of-time compiler" on page 25
Ahead-of-Time (AOT) compilation allows you to compile Java classes and methods before you run your code. AOT compilation avoids the unpredictable timing effect the JIT compiler can have on sensitive performance paths. To ensure your code is compiled before it executes and for the highest level of deterministic performance, you can precompile your code into a shared class cache using the AOT compiler.

Related reference

"Ahead-of-time options" on page 329
The definitions for the ahead-of-time options.

The Just-In-Time (JIT) compiler

You can control when and how the JIT compiler operates using the java.lang.Compiler class that is provided as part of the standard SDK class library. IBM fully supports the Compile.compileClass(), Compiler.enable() and Compiler.disable() methods.

For example, if you want to warm up your application and know that the key methods in your application have been compiled, you can call the Compiler.disable() method after you have warmed up your application and be confident that JIT compilation will not occur during the remainder of the execution of your application.

You can control method compilation in two ways:

- Specify a set of methods that you can compile:

  ```
  Compiler.command("{<method specification>}(compile)");
  ```

  where <*method specification*> is a list of all the methods that have been loaded at this point and are to be compiled. <*method specification*> describes a fully qualified method name. An asterisk denotes a wildcard match.

  For example, to compile all methods starting with java.lang.String that were already loaded, you specify:

  ```
  Compiler.command("{java.lang.String*}(compile)");
  ```

  Note: This command compiles not only methods in the java.lang.String class, but also in the java.lang.StringBuffer class, which might not be what you wanted. To compile only methods in the java.lang.String class, you specify:

  ```
  Compiler.command("{java.lang.String.*}(compile)");
  ```

- Specify that all methods in the compilation queue will be compiled before this thread runs and continues:

  ```
  Compiler.command("waitOnCompilationQueue");
  ```

  You might want to ensure that the compilation queue was empty before disabling the compiler. A typical technique for compiling a set of methods and classes might be:

```
Compiler.enable();                              // ensure compiler is active
Compiler.command("{com.mycompany.*}(compile)"); // queue up all the methods you want to compile
Compiler.command("waitOnCompilationQueue");     // wait until all those methods are compiled
Compiler.disable();                             // turn the compiler off
```

Determinism during JNI transitions

By default, the JIT generates optimized code for high performance Java-to-native JNI transitions. Reduced determinism might possibly occur when reloading a native library using the following code sequence:

```
RegisterNatives / UnregisterNatives / RegisterNatives
```

To revert to the slower, more deterministic code, use the command line option **-Xjit:disableDirectToJNI**.

Enabling the JIT

You can explicitly enable the JIT in several ways. Both command-line options override the **JAVA_COMPILER** environment variable.

Procedure

- Set the **JAVA_COMPILER** environment variable to "jitc" before running the Java application. At a shell prompt, enter:
  - **For the Korn shell:** `export JAVA_COMPILER=jitc`

    Note: Korn shell commands are used in this information unless otherwise stated.
  - **For the Bourne shell:**

    ```
    JAVA_COMPILER=jitc
    export JAVA_COMPILER
    ```
  - **For the C shell:** `setenv JAVA_COMPILER jitc`

  If the **JAVA_COMPILER** environment variable is an empty string, the JIT remains disabled. To disable the environment variable, at a shell prompt, enter `unset` **`JAVA_COMPILER`**.
- Use the **-D** option on the JVM command line to set the java.compiler property to "jitc". At a shell prompt, enter: `java -Djava.compiler=jitc <MyApp>`
- Use the **-Xjit** option on the JVM command line. You must *not* specify the **-Xint** option at the same time. At a shell prompt, enter: `java -Xjit <MyApp>`

Disabling the JIT

You can disable the JIT in several ways. Both command-line options override the **JAVA_COMPILER** environment variable.

About this task

Procedure

- Set the **JAVA_COMPILER** environment variable to "NONE" or the empty string before running the Java application. At a shell prompt enter:
  - **For the Korn shell:** `export JAVA_COMPILER=NONE`

    Note: Korn shell commands are used for the remainder of this information.
  - **For the Bourne shell:**

    ```
    JAVA_COMPILER=NONE
    export JAVA_COMPILER
    ```
  - **For the C shell:** `setenv JAVA_COMPILER NONE`
- Use the **-D** option on the JVM command line to set the java.compiler property to "NONE" or the empty string. At a shell prompt, enter: `java -Djava.compiler=NONE <MyApp>`

- Use the **-Xint** option on the JVM command line. At a shell prompt, enter: `java -Xint <MyApp>`

Determining whether the JIT is enabled

You can determine the status of the JIT using the **-version** option.

Procedure

Enter the following at a shell prompt:

```
java -version
```

If the JIT is not in use, a message is displayed that includes the following:
`(JIT disabled)`
If the JIT is in use, a message is displayed that includes the following:
`(JIT enabled)`

Chapter 5. Class data sharing between JVMs

The Java Virtual Machine (JVM) allows you to share class data between JVMs by storing it in a memory-mapped cache file on disk.

Sharing reduces the overall virtual storage consumption when more than one JVM shares a cache. Sharing also reduces the startup time for a JVM after the cache has been created. The shared class cache is independent of any active JVM and persists until it is destroyed. A shared cache can contain:

- Bootstrap classes
- Application classes
- Metadata that describes the classes
- Ahead-of-time (AOT) compiled code

Shared class caches can be used by IBM WebSphere Real Time for RT Linux in both non-real-time and real-time modes, but the cache format, creation, and population techniques differ. Real-time mode caches are not compatible with non-real-time mode caches. In non-real-time mode, caches are created and populated in the same way as the standard JVM. This means the cache is created and populated by the JVM as it runs an application, transparently to the user. In real-time mode, using the **-Xrealtime** option, shared class caches must be created and prepopulated by admincache, using the **-populate** option. Applications running in real-time mode might read content from the prepopulated cache, but cannot modify its contents.

Use the admincache tool to create, populate, and destroy caches.

To enable an application to use a shared class cache, add the **-Xshareclasses** option to its command-line. Because real-time mode caches are read-only, some non-real-time mode suboptions of **-Xshareclasses** are not available in real-time mode.

For further information, see Using the admincache tool, Class data sharing between JVMs for non Real-Time, and Shared classes diagnostics.

Running Applications with a Shared Class Cache

To run an application with a shared class cache, use the **-Xshareclasses** option on the command-line.

Table 8 shows the suboptions available when running an application in real-time mode, using the **-Xshareclasses** option.

Table 8. Suboptions available when running an application in real-time mode

Option	Meaning
`cacheDir=<directory>`	Sets the directory in which shared class cache data is read and written. By default, *<directory>* is `/tmp/javasharedresources`. The directory name must match that specified in the **-cacheDir** option used in the admincache command to create the cache.

Table 8. Suboptions available when running an application in real-time mode (continued)

Option	Meaning
`name=<name>`	The name of the shared class cache to use. The name must match that specified in the **-cacheName** option used in the admincache command to create the cache. The name must not exceed 53 characters.
`none`	Explicitly disable class sharing. Can be added to the end of a command line to disable class data sharing. This suboption overrides class sharing arguments found earlier on the command line.
`nonfatal`	Always start JVM regardless of errors or warnings. Allows the JVM to start even if class data sharing fails. Typical behavior for the JVM is to refuse to start if class data sharing fails. If you select **nonfatal** and the shared classes cache fails to initialize, the JVM attempts to connect to the cache in read-only mode. If this attempt fails, the JVM starts without class data sharing.
`silent`	Suppress all output messages. Turns off all shared classes messages, including error messages. Unrecoverable error messages, which prevent the JVM from initializing, are displayed.
`verbose`	Enable verbose output, providing overall status on the shared class cache and more detailed error messages.
`verboseAOT`	Enables verbose output when compiled AOT code is being found in the cache, for example during AOT method load requests.
`verboseHelper`	Enables verbose output for the Java Helper API. This output shows you how the Helper API is used by your ClassLoader.
`verboseIO`	Enable verbose output of class load requests. This option provides detailed output about the cache I/O activity, listing information about classes being found.

To ensure these options are correct, use the **-printvmargs** option with admincache (see for more information). The **nonfatal** option is not suitable for general use, because it forces the JVM to ignore warnings and errors about the shared class cache. The **none** option explicitly disables class sharing, and is equivalent to omitting the **-Xshareclasses** option on the command line.

For more detailed information about the **-Xshareclasses** suboptions, see “Class data sharing command-line options” on page 400.

Chapter 6. Using the Metronome Garbage Collector

Metronome Garbage Collector replaces the standard Garbage Collector in WebSphere Real Time for RT Linux.

Related reference

"Metronome Garbage Collector options" on page 336
The definitions of the Metronome Garbage Collector options.

Introduction to the Metronome Garbage Collector

The benefit of the Metronome Garbage Collector is that the time it takes is more predictable and garbage collection can take place at set intervals over a period of time.

The key difference between Metronome garbage collection and standard garbage collection is that Metronome garbage collection occurs in small interruptible steps but standard garbage collection stops the application while it marks and collects garbage.

You can control garbage collection with the Metronome Garbage Collector using the **-Xgc:targetUtilization=N** option to limit the amount of CPU used by the Garbage Collector.

For example:

```
java -Xrealtime -Xgc:targetUtilization=80 yourApplication
```

The example specifies that your application runs for 80% in every 60ms. The remaining 20% of the time is used for garbage collection. The Metronome Garbage Collector guarantees utilization levels provided that it has been given sufficient resources. Garbage collection begins when the amount of free space in the heap falls below a dynamically determined threshold.

Garbage collection and priorities

The garbage collection thread has to run at a priority higher than the highest priority thread that generates garbage in the heap; otherwise, it might not run as specified by the configured utilization. Both regular Java threads and real-time threads can generate garbage and, therefore, garbage collection must run at a priority higher than all regular and real-time threads. This prioritization is handled automatically by the JVM and garbage collection runs at 0.5 priority above the highest priority of all regular and real-time threads. However, it is important to ensure that no-heap real-time threads (NHRTs) are not affected by garbage collection. Run all NHRTs at a higher priority than the highest priority real-time threads. This means that NHRTs run at a priority higher than garbage collection and are not delayed.

Table 9 on page 50 shows a typical example of priorities that you can define and the related garbage collection priorities that follow from your choice

For the comparison of Java priorities and OS priorities, see "Priority mapping and inheritance" on page 63.

Table 9. Example of garbage collection and priorities

Threads	Priorities (examples)
If the highest priority real-time thread is:	20 (OS priority 43)
Then the Garbage Collector is:	20.5 (OS priority 44)
To ensure that a NHRT runs independently of the garbage collector, set a higher priority than the GC:	21 (OS priority 45) or higher.
The Metronome alarm thread is:	Priority 46 (OS priority 89)

Note: Even with this configuration, no-heap real-time threads are not completely unaffected by garbage collection because the metronome alarm thread runs at the highest priority in the system to ensure that it can wake up regularly and work out if garbage collection needs to do anything. The work to do that is, of course, tiny and thus not a major consideration.

Metronome garbage collection and class unloading

The Metronome Garbage Collector does not unload classes in IBM WebSphere Real Time because it can require a non-deterministic amount of work causing pause time outliers.

Metronome Garbage Collector threads

The Metronome Garbage Collector consists of two types of threads: a single alarm thread, and a number of collection (GC) threads. By default, there is one GC thread. You can set the number of GC threads for the JVM using the **-Xgcthreads** option.

You cannot change the number of alarm threads for the JVM.

The Metronome Garbage Collector periodically checks the JVM to see if the heap memory has sufficient free space. When the amount of free space falls below the limit, the Metronome Garbage Collector triggers the JVM to start garbage collection.

Alarm thread

The single alarm thread guarantees to use minimal resources. It "wakes" at regular intervals and makes these checks:

- The amount of free space in the heap memory
- Whether garbage collection is currently taking place

If insufficient free space is available and no garbage collection is taking place, the alarm thread triggers the collection threads to start garbage collection. The alarm thread does nothing until the next scheduled time for it to check the JVM.

Collection threads

Each collection thread checks Java and real-time threads for heap objects. They check the memory areas in the following sequence:

1. Scoped memory to identify and mark any live objects in the heap that are being used by objects from scoped memory.
2. Immortal memory to identify and mark any live objects in the heap that are being used by objects from immortal memory.
3. Heap memory to identify and mark live objects.

When the live objects have been marked, the unmarked objects are available for collection.

After the garbage collection cycle has completed, the Metronome Garbage Collector checks the amount of free heap space. If there is still insufficient free heap space, another garbage collection cycle is started using the same trigger id. If there is sufficient free heap space, the trigger ends and the garbage collection threads are stopped. The alarm thread continues to monitor the free heap space and will trigger another garbage collection cycle when it is required.

Related concepts

"Memory management" on page 65
Garbage collecting memory heaps has always been considered an obstacle to real-time programming because of the unpredictable behavior introduced by garbage collection. The Metronome garbage collector in IBM WebSphere Real Time for RT Linux can provide high deterministic GC performance. At the same time, the Real-Time Specification for Java (RTSJ) provides several extensions to the memory model for objects outside the garbage-collected heap, so that a Java programmer can explicitly manage both short-lived and long-lived objects.

"Priority mapping and inheritance" on page 63
Each Java priority is mapped to an associated operating system base priority, and each operating system priority is associated with a scheduling policy. The WebSphere Real Time for RT Linux Linux operating system scheduling policies are SCHED_OTHER, SCHED_RR and SCHED_FIFO.

Related tasks

"Planning your WebSphere Real Time for RT Linux application" on page 88
When you are preparing to write real-time Java applications, you must consider whether to use Java threads, real-time threads, or no-heap real-time threads. In addition, you can decide which memory area that your threads will use.

Related reference

"Metronome Garbage Collector options" on page 336
The definitions of the Metronome Garbage Collector options.

Troubleshooting the Metronome Garbage Collector

Using the command-line options, you can control the frequency of Metronome garbage collection, out of memory exceptions, and the Metronome behavior on explicit system calls.

Related concepts

Chapter 12, "Troubleshooting OutOfMemory Errors," on page 109
Dealing with OutOfMemoryError exceptions, memory leaks, and hidden memory allocations

Related information

"Tracing Java applications and the JVM" on page 208
JVM trace is a trace facility that is provided in all IBM-supplied JVMs with minimal affect on performance. In most cases, the trace data is kept in a compact binary format, that can be formatted with the Java formatter that is supplied.

Using verbose:gc information

You can use the **-verbose:gc** option with the **-Xgc:verboseGCCycleTime=N** option to write information to the console about Metronome Garbage Collector activity. Not all XML properties in the **-verbose:gc** output from the standard JVM are created or apply to the output of Metronome Garbage Collector.

Use the **-verbose:gc** option to view the minimum, maximum, and mean free space in the heap. In this way, you can check the level of activity and use of the heap and subsequently adjust the values if necessary. The **-verbose:gc** option writes Metronome statistics to the console.

The **-Xgc:verboseGCCycleTime=N** option controls the frequency of retrieval of the information. It determines the time in milliseconds that the summaries are dumped. The default value for N is 1000 milliseconds. The cycle time does not mean the summary is dumped precisely at that time, but when the last garbage collection event that meets this time criterion passes. The collection and display of these statistics can distort Metronome Garbage Collector pause time targets and, as N gets smaller, the distortion can become quite large.

A quantum is a single period of Metronome Garbage Collector activity, causing an interruption or pause time for an application.

Example of verbose:gc output

Enter:

```
java -Xrealtime -verbose:gc -Xgc:verboseGCCycleTime=N myApplication
```

This example shows the initial output of verbose:gc, which contains the version and garbage collection settings:

```
<verbosegc version="20081016_AA">

<initialized>
  <attribute name="gcPolicy" value="-Xgcpolicy:metronome" />
  <attribute name="maxHeapSize" value="0x46800000" />
  <attribute name="initialHeapSize" value="0x38400000" />
  <attribute name="gcthreads" value="1" />
  <attribute name="compressedPointersDisplacement" value="0x0" />
  <attribute name="compressedPointersShift" value="0x0" />
  <attribute name="beatsPerMeasure" value="500" />
  <attribute name="timeInterval" value="10000" />
  <attribute name="targetUtilization" value="50" />
  <attribute name="trigger" value="0x1c200000" />
  <attribute name="headRoom" value="0x100000" />
  <attribute name="pageSize" value="0x4000" />
</initialized>
```

When garbage collection is triggered, a trigger start event occurs, followed by any number of heartbeat events, then a trigger end event when the trigger is satisfied. This example shows a triggered garbage collection cycle as verbose:gc output:

```
<gc type="trigger start" id="5" timestamp="Oct 21 12:16:55 2008" intervalms="231.498" />

<gc type="heartbeat" id="5" timestamp="Oct 21 12:16:55 2008" intervalms="322.118">
  <summary quantumcount="157">
    <quantum minms="0.013" meanms="0.476" maxms="0.647" />
    <exclusiveaccess minms="0.006" meanms="0.008" maxms="0.038" />
    <refs_cleared soft="1" threshold="24" maxThreshold="32" weak="0" phantom="0" />
    <heap minfree="32789532" meanfree="76387033" maxfree="167371268" />
    <immortal minfree="16057876" meanfree="16057876" maxfree="16057876" />
    <gcthreadpriority max="11" min="11" />
  </summary>
</gc>

<gc type="trigger end" id="5" timestamp="Oct 21 12:16:55 2008" intervalms="322.140" />
```

The following event types can occur:

<gc type="trigger start" ...>
The start of a garbage collection cycle. The used memory became higher than the trigger threshold. The default threshold is 50% of heap. You can change the threshold by using the -XXgc:trigger=*NN*, where *NN* is an absolute amount of memory. The intervalms attribute is the interval between the previous trigger end event (with id-1) and this trigger start event.

<gc type="trigger end" ...>
A garbage collection cycle successfully lowered the amount of used memory to below the trigger threshold. If a garbage collection cycle ended, but used memory did not drop below the trigger threshold, a new garbage collection cycle is started as part of the same trigger ID. For each trigger start event, there is a matching trigger end event with same ID. The intervalms attribute is the interval between the previous trigger start event (with the same id) and the current trigger end event. During this period of time, one or more garbage collection cycles will have completed until used memory has dropped below the trigger threshold.

<gc type="heartbeat" ...>
A periodic event that gathers information (on memory and time) about all garbage collection quanta for the period of time it covers. A heartbeat event can occur only between a matching pair of triggerstart and triggerend events; that is, while an active garbage collection cycle is in process. The intervalms attribute is the interval between the previous heartbeat event (with id -1) and this heartbeat event.

<gc type="syncgc" ...>
A synchronous (nondeterministic) garbage collection event. See "Synchronous garbage collections" on page 54

The XML tags in this example have the following meanings:

<summary ...>
A summary of the garbage collection activity during the heartbeat interval. The quantumcount attribute is the number of garbage collection quanta run in the summary period.

<quantum ...>
A summary of the length of quantum pause times during the heartbeat interval in milliseconds.

<heap ...>
A summary of the amount of free heap space during the heartbeat interval, sampled at the end of each garbage collection quantum.

<immortal ...>
A summary of the amount of free space in the immortal heap during the heartbeat interval, sampled at the end of each garbage collection quantum.

<refs_cleared ...>
Is the number of Java reference objects that were cleared during the heartbeat interval.

<gcthreadpriority ...>
The maximum priority and minimum priority of garbage collection threads during the heartbeat interval.

Note:

- If only one garbage collection quantum occurred in the interval between two heartbeats, the free memory is sampled only at the end of this one quantum, and therefore the minimum, maximum, and mean amounts given in the heartbeat summary are all equal.
- It is possible that the interval might be significantly larger than the cycle time specified because the garbage collection has no work on a heap that is not full enough to warrant garbage collection activity. For example, if your program requires garbage collection activity only once every few seconds, you are likely to see a heartbeat only once every few seconds.

  If an event such as a synchronous garbage collection or a priority change occurs, the details of the event and any pending events, such as heartbeats, will be immediately produced as output.
- If the maximum garbage collection quantum for a given period is too large, you might want to reduce the target utilization using the **-Xgc:targetUtilization** option to give the Garbage Collector more time to work, or you might want to increase the heap size using the **-Xmx** option. Similarly, if your application can tolerate longer delays than are currently being reported, you can increase the target utilization or decrease the heap size.
- The output can be redirected to a log file instead of the console using the **-Xverbosegclog:<file>** option; for example, **-Xverbosegclog:out** writes the **-verbose:gc** output to the file *out*.
- The priority listed in gcthreadpriority is the underlying OS thread priority, not a Java thread priority.

Synchronous garbage collections

An entry is also written to the **-verbose:gc** log when a synchronous (nondeterministic) garbage collection occurs. This event has three possible causes:

- An explicit `System.gc()` call in the code.
- The JVM running out of memory and performing a synchronous garbage collection to avoid an OutOfMemoryErrorcondition.
- The JVM shutting down, while there is a continuous garbage collection. The JVM cannot just cancel that collection, but finishes it synchronously and only then exits.

An example of a `System.gc()` entry is:

```
<gc type="synchgc" id="1" timestamp="Oct 21 12:39:39 2008" intervalms="4.184">
  <details reason="system garbage collect" />
  <duration timems="40.149" />
  <finalization objectsqueued="82" />
  <heap freebytesbefore="921001248" />
  <heap freebytesafter="940591184" />
  <immortal freebytesbefore="101550472" />
  <immortal freebytesafter="101550472" />
  <synchronousgcpriority value="11" />
</gc>
```

This example shows a synchronous garbage collection entry as a result of an out-of-memory condition, and the events leading to it:

```
<gc type="trigger start" id="1" timestamp="Oct 21 12:13:42 2008" intervalms="0.000" />

<gc type="heartbeat" id="1" timestamp="Oct 21 12:13:43 2008" intervalms="280.857">
  <summary quantumcount="138">
    <quantum minms="0.013" meanms="0.475" maxms="0.554" />
    <exclusiveaccess minms="0.006" meanms="0.011" maxms="0.064" />
    <refs_cleared soft="0" threshold="9" maxThreshold="32" weak="1" phantom="0" />
```

```
    <finalization objectsqueued="36" />
    <heap minfree="682380" meanfree="25635229" maxfree="52210096" />
    <immortal minfree="16119944" meanfree="16119944" maxfree="16119944" />
    <gcthreadpriority max="11" min="11" />
  </summary>
</gc>

<gc type="synchgc" id="1" timestamp="Oct 21 12:13:43 2008" intervalms="0.245">
  <details reason="out of memory" />
  <duration timems="6.015" />
  <heap freebytesbefore="550204" />
  <heap freebytesafter="550204" />
  <immortal freebytesbefore="16119944" />
  <immortal freebytesafter="16119944" />
  <synchronousgcpriority value="11" />
</gc>
```

An example of a synchronous garbage collection entry as a result of JVM shutting down is:

```
<gc type="synchgc" id="27" timestamp="Mar 01 09:31:30 2007" intervalms="2.527">
  <details reason="vm shutdown" />
  <duration timems="43.090" />
  <heap freebytesbefore="30031872" />
  <heap freebytesafter="57851904" />
  <immortal freebytesbefore="16119944" />
  <immortal freebytesafter="16119944" />
  <synchronousgcpriority value="11" />
</gc>
```

The XML tags and attributes in this example have the following meanings:

<gc type="syncgc" ...>
: Is the event specifying that this is a synchronous garbage collection. The intervalms attribute is the interval between previous event (heartbeat, trigger start, trigger end, or another synchgc) and the beginning of this synchronous garbage collection.

<details ...>
: Is the cause of the synchronous garbage collection.

<duration ...>
: Is the time it took to complete this garbage collection cycle synchronously in milliseconds.

<heap ...>
: Is the free Java heap memory before and after the synchronous garbage collection in bytes.

<immortal ...>
: Is the free immortal heap memory before and after the synchronous garbage collection in bytes.

<synchronousgcpriority ...>
: Is the priority of garbage collection threads during the synchronous garbage collection. This is an operating system thread priority, not a Java thread priority.

<finalization ...>
: Is the number of objects awaiting finalization.

Synchronous garbage collection due to out-of-memory conditions or VM shut down can happen only when the Garbage Collector is active. It has to be preceded by a trigger start event, although not necessarily immediately. Some heartbeat

events probably occur between a trigger start event and the synchgc event. Synchronous garbage collection caused by `System.gc()` can happen at any time.

Tracking all GC quanta

Individual GC quanta can be tracked by enabling the **Global GC Start** and **Global GC End** tracepoints. These tracepoints are produced at the beginning and end of all Metronome Garbage Collector activity including synchronous garbage collections. The output for these tracepoints will look similar to:

```
03:44:35.281 0x833cd00 j9mm.52 - GlobalGC start: weakrefs=7 soft=11 phantom=0 finalizers=75 globalcou

03:44:35.284 0x833cd00 j9mm.91 - GlobalGC end: workstackoverflow=0 overflowcount=0 weakrefs=7 soft=1
```

Priority changes

In addition to summaries, an entry is written to the **-verbose:gc** log when the Garbage Collector thread priority changes (as a result of the application changing thread priorities, or one or more threads in an application ending). The priority listed is the underlying OS thread priority, not a Java thread priority. An example of a Garbage Collector thread priority change entry is:

```
<gc type="heartbeat" id="73" timestamp="Feb 26 13:11:35 2007" intervalms"1001.754">
  <summary quantumcount="240">
    <quantum minms="0.022" meanms="0.984" maxms="1.011" />
    <classunloading classloaders="11" classes="17" />
    <heap minfree="202833920" meanfree="214184823" maxfree="221102080" />
    <gcthreadpriority max="11" min="11" />
  </summary>
</gc>
```

Priority changes can be tracked in Real Time by producing the trace point information relating to Garbage Collector thread priorities. This output looks similar to:

```
15:58:25.493*0x8286e00    j9mm.102       - setGCThreadPriority() called with newGCThreadPriority = 1
```

This output can be enabled by using its ID, as follows:
-Xtrace:iprint=tpnid{j9mm.102}

Out-of-memory entries

When one of the memory areas runs out of free space, an entry is written to the **-verbose:gc** log before the OutOfMemoryError exception is thrown. An example of this output is:

```
<event details="out of memory" timestamp="Aug 03 10:49:18 2006" memoryspace="Scoped" J9MemorySpace="0x08482F30" />
```

By default a Javadump is produced as a result of an OutOfMemoryError exception. This dump contains information about the memory areas used by your program. Together with the `J9MemorySpace` value given in the **-verbose:gc** output, you can use this information in the dump to identify the particular memory area that ran out of space:

```
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
<< output removed for clarity >>
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Scoped Segment  ID=08482F30
1STSEGMENT     08482780 08807C48 08816748 08816748  00002008 eb00
1STSEGSUBTYPE  Immortal Segment  ID=08482F14
```

```
1STSEGMENT     08482708 B26EA008 B36EA008 B36EA008  00001008 1000000
1STSEGSUBTYPE  Heap Segment  ID=08482EF8
1STSEGMENT     08482690 B36EB008 B76EB008 B76EB008  00000009 4000000
NULL
```

In the example above, the memory space ID given in the **-verbose:gc** output (`0x08482F30`) can be matched to the ID of the Scoped Segment memory area of the Javadump. This match can be useful if you have several scopes and need to identify which one has gone out of memory, because the **-verbose:gc** output only indicates whether the OutOfMemoryError occurred in immortal, scoped, or heap memory, rather than down to the level of individual scopes.

Related reference

"Metronome Garbage Collector options" on page 336
The definitions of the Metronome Garbage Collector options.

Metronome Garbage Collector behavior in out-of-memory conditions

By default, the Metronome Garbage Collector triggers an unlimited, nondeterministic garbage collection when the JVM runs out of memory. To prevent nondeterministic behavior, use the **-Xgc:noSynchronousGCOnOOM** option to throw an OutOfMemoryError when the JVM runs out of memory.

The default unlimited collection runs until all possible garbage is collected in a single operation. The pause time required is usually many milliseconds greater than a normal metronome incremental quantum.

Related information

Using -Xverbose:gc to analyze synchronous garbage collections

Metronome Garbage Collector behavior on explicit System.gc() calls

If a garbage collection cycle is in progress, the Metronome Garbage Collector completes the cycle in a synchronous way when `System.gc()` is called. If no garbage collection cycle is in progress, a full synchronous cycle is performed when `System.gc()` is called. Use `System.gc()` to clean up the heap in a controlled manner. It is a nondeterministic operation because it performs a complete garbage collection before returning.

Some applications call vendor software that has System.gc() calls where it is not acceptable to create these nondeterministic delays. To disable all System.gc() calls use the **-Xdisableexplicitgc** option.

The verbose garbage collection output for a System.gc() call has a reason of "system garbage collect" and is likely to have a long duration:

```
<gc type="synchgc" id="1" timestamp="Oct 21 12:39:39 2008" intervalms="4.184">
  <details reason="system garbage collect" />
  <duration timems="40.149" />
  <finalization objectsqueued="82" />
  <heap freebytesbefore="921001248" />
  <heap freebytesafter="940591184" />
  <immortal freebytesbefore="101550472" />
  <immortal freebytesafter="101550472" />
  <synchronousgcpriority value="11" />
</gc>
```

Metronome Garbage Collector limitation

When using the Metronome Garbage Collector, you might experience longer than expected pauses during garbage collection.

During garbage collection, a root scanning process is used. The garbage collector walks the heap, starting at known live references. These references include:

- Live reference variables in the active thread call stacks.
- Static references.
- All object references in immortal and scoped memories.

To find all the live object references on an application thread's stack, the garbage collector scans all the stack frames in that thread's call stack. Each active thread stack is scanned in an uninterruptible step. This means the scan must take place within a single GC quantum.

The effect is that the system performance might be worse than expected if you have some threads with very deep stacks, because of extended garbage collection pauses at the beginning of a collection cycle.

Immortal memory is processed incrementally. All other scoped memory areas are processed in one atomic, uninterruptible step. This means that significant use of scoped memory areas might lead to worse system performance than expected, because of extended garbage collection pauses when the root scan is processing scoped memory.

Tuning Metronome Garbage Collector

You can tune the real-time environment by controlling the amount of memory that your application uses. For example, use the **-Xmx**, **-Xgc:immortalMemorySize=size**, **-Xgc:scopedMemoryMaximumSize=size**, and the **-Xgc:targetUtilization=N** options.

- Use the **-Xmx** option to limit the size of the heap.

  The value chosen is used as the upper limit of heap size and thus reflects the likely usage over time. Choosing a value that is too low increases the garbage collection frequency and leads to a lower overall throughput although it reduces the memory footprint. For good real-time performance, avoid paging. It is normal to ensure that the footprint of all the running processes on a machine does not exceed the physical memory size.
- Use the **-Xgc:immortalMemorySize=size** option to control the size of the immortal memory area.

  You must analyze carefully the use of immortal memory. The “ideal” application uses immortal memory during startup but thereafter stops using it. If allocation of immortal objects continues, the application is able to continue to run until immortal memory has been exhausted. The current usage can be obtained by adding:

  ```
  long used = ImmortalMemory.instance().memoryConsumed();
  ```

  to your code.
- Use the **-Xgc:scopedMemoryMaximumSize=size** option to ensure that applications do not request excessive amounts of scoped memory. Use this option for diagnosis rather than tuning.

- Set the **-Xgc:targetUtilization=N** option to ensure that under the worst-case conditions (maximum allocation rate of heap objects), the garbage collector can collect garbage at a higher rate than the application generates it.

  Typically, the default value is sufficient but application performance might be improved by increasing the utilization to the point at which the collector is able to collect garbage slightly faster than the application can create it.
- Use the **-Xgcthreads <n>** option to create additional threads to run garbage collection in parallel.

  The default is to use one thread. If your workload has a high rate of garbage generation, and runs on a symmetric multiprocessor with CPU cycles available, performance could benefit by setting this parameter to >1.

  Note: Setting this parameter too high can have a negative affect on throughput.

Chapter 7. Support for RTSJ

WebSphere Real Time for RT Linux implements the Real-Time Specification for Java (RTSJ).

WebSphere Real Time for RT Linux version 2.0 has been certified as RTSJ compliant against the RTSJ Technology Compatibility Kit 1.0.2 version J9 3.0.13 FCS and is compliant with the Java Compatibility Kit (JCK) version 6.0.

Related reference

"WebSphere Real Time for RT Linux class libraries" on page 346
A reference to the Java class libraries that are used by WebSphere Real Time for RT Linux.

Real-time thread scheduling and dispatching

Thread scheduling and dispatching of real-time Java threads is part of the Real Time Specification for Java. The scheduling policy SCHED_FIFO is used to prioritize real-time Java threads using Linux operating system priorities 11 - 89.

Information about Linux scheduling policies can be found in Chapter 3, "Thread scheduling and dispatching," on page 17.

Related concepts

"Launching secondary processes" on page 20
The java.lang.Runtime.exec methods in the Java virtual machine (JVM) API give your Java application the ability to execute a command in a separate process.

Schedulables and their Parameters

There are two main types of real-time schedulable objects: real-time threads and asynchronous event handlers.

These schedulable objects have the following parameters associated with them:

SchedulingParameters
PriorityParameters schedules real-time schedulable objects by priority.

ReleaseParameters

- **PeriodicParameters** describes periodic release of real-time schedulable objects. A periodic real-time thread is one that is released at regular intervals.
- **AperiodicParameters** describes the release of real-time schedulable objects. Aperiodic real-time threads are released at irregular intervals.

MemoryParameters
Describes memory allocation constraints for real-time schedulable objects.

ProcessingGroupParameters
Unsupported in WebSphere Real Time for RT Linux.

The priority scheduler

In WebSphere Real Time for RT Linux, the scheduler is a priority scheduler. As its name implies, it manages the running of schedulable objects according to their active priorities.

The scheduler maintains the list of schedulable objects and determines when each object can be released to run in the CPU. The scheduler must abide by the various parameters that are associated with each schedulable object. The methods addToFeasibility, isFeasible, and removeFromFeasibility are provided for this purpose.

Priorities and policies

Regular Java threads, that is, threads allocated as `java.lang.Thread` objects, can use scheduling policies SCHED_OTHER, SCHED_RR or SCHED_FIFO. Real-time threads, that is, threads allocated as `java.lang.RealtimeThread`, and asynchronous event handlers use the SCHED_FIFO scheduling policy.

Regular Java threads use the default scheduling policy of SCHED_OTHER, unless the JVM is started by a thread with policy SCHED_RR or SCHED_FIFO. Regular Java threads that use the policy SCHED_OTHER have the operating system thread priority set to 0. Regular Java threads that use the policy SCHED_RR or SCHED_FIFO inherit the priority of the thread that starts the JVM. For more information about priorities and policies for regular Java threads, see "Regular Java thread priorities and policies" on page 18.

For real-time threads, the SCHED_FIFO policy has no time slicing and supports 99 priorities from 1 (the lowest) to 99 (the highest). This WebSphere Real Time for RT Linux implementation supports 28 user priorities in the range 11-38 inclusive, so:

```
javax.realtime.PriorityScheduler().getMinPriority()
```

returns 11, and:

```
javax.realtime.PriorityScheduler().getMaxPriority()
```

returns 38.

OS priorities 81 - 89 are used by the IBM JVM for dispatching worker threads. These threads are all designed to do a small amount of work before going back to sleep. The threads are as follows:

- The Metronome Garbage Collector alarm thread runs at a priority of 89. This thread runs regularly and dispatches a GC work unit.
- Two Asynchronous Signal Threads, which process asynchronous signals, one being a no-heap real-time (NHRT) thread at priority 88 and the other at priority 87.
- Two Timer Threads, which dispatch timer events, one being a no-heap real-time thread for no-heap timers at priority 85 and the other at priority 83.
- The Async Event Handler Threads, which are dispatched to run asynchronous event handlers, and, while running an async event handler, are assigned the priority of that handler. The system starts up with two no-heap real-time handler threads at priority 85 and 8 others at priority 83.
- The Asynchronous Signal no-heap real-time thread at priority 88 handles requests for heap dumps, core dumps, and javacore dumps. It temporarily boosts its priority to 89 while creating dump files.

The Metronome GC Trace thread runs at OS priority 12, and the JIT Sampler thread, which samples Java methods for compilation, runs at OS priority 13.

The JIT Compilation thread (which is different from the JIT Sampler thread) runs with the SCHED_OTHER policy at OS priority 0.

The JIT compilation and JIT sampler threads are both disabled if **-Xnojit** or **-Xint** is specified.

The Metronome Garbage Collector and finalizer priority constantly changes (before each round of collection) to be above the highest priority heap-allocating thread. You must ensure that the priority of heap-allocating threads is below that of NoHeapRealtimeThreads.

A heap-allocating thread is any non-NHRT user thread that is not asleep or blocked on a monitor. A user thread running native code outside the JNI interface is not considered to be heap-allocating. If a garbage collection is in progress when a heap-allocating thread wakes up, is no longer blocked on a monitor, or leaves JNI, it is forced to wait until the garbage collection has finished before it can continue.

OS priority 81 is reserved for internal JVM threads that are allocating from the heap. If an internal JVM thread is at OS priority 81, the garbage collector runs at OS priority 82. When the only heap-allocating user threads are not real-time threads, the GC priority runs at OS priority 11. Otherwise, the GC runs at a priority that is one OS priority higher than the highest priority heap-allocating user thread.

The GC priority is adjusted just before a round of collection.

Related concepts

"Launching secondary processes" on page 20
The java.lang.Runtime.exec methods in the Java virtual machine (JVM) API give your Java application the ability to execute a command in a separate process.

Related tasks

"Writing real-time threads" on page 90
So far, you have just modified an application; now it is time to write some code. You can write applications that use real-time threads to take advantage of the real-time priority levels and memory areas.

Priority mapping and inheritance

Each Java priority is mapped to an associated operating system base priority, and each operating system priority is associated with a scheduling policy. The WebSphere Real Time for RT Linux Linux operating system scheduling policies are SCHED_OTHER, SCHED_RR and SCHED_FIFO.

Real-time Java threads use policy SCHED_FIFO, while regular Java threads use the policy of the thread that starts the JVM. The default scheduling policy for regular Java threads is SCHED_OTHER, but you can use a utility like chrt to set policies SCHED_RR or SCHED_FIFO. For more information about thread priorities and policies, see Chapter 3, "Thread scheduling and dispatching," on page 17.

The following table shows how the Java priorities are mapped to native operating system priorities. Some Java priorities are reserved for use by the JVM, and some native priorities that have no corresponding Java priorities are used by the JVM as well.

Note:

- Priorities 1-10 are used by regular Java threads.
  - For policy SCHED_OTHER, Java priorities 1-10 map to operating system priority 0.

  - For policies SCHED_FIFO or SCHED_RR, Java priorities 1-10 inherit the priority of the thread that starts the JVM.
- Priorities 11 upwards are used by real-time threads and no-heap real-time threads
- A schedulable object always runs with its active priority. The active priority is initially the base priority of the schedulable object, but the active priority can be temporarily raised by priority inheritance. The base priority of a schedulable object can be changed while it is running.

User base priorities:

```
Java priorities 1-10: SCHED_OTHER, OS priority 0

Java priority 11: SCHED_FIFO, OS priority 25
Java priority 12: SCHED_FIFO, OS priority 27
Java priority 13: SCHED_FIFO, OS priority 29
Java priority 14: SCHED_FIFO, OS priority 31
Java priority 15: SCHED_FIFO, OS priority 33
Java priority 16: SCHED_FIFO, OS priority 35
Java priority 17: SCHED_FIFO, OS priority 37
Java priority 18: SCHED_FIFO, OS priority 39
Java priority 19: SCHED_FIFO, OS priority 41
Java priority 20: SCHED_FIFO, OS priority 43
Java priority 21: SCHED_FIFO, OS priority 45
Java priority 22: SCHED_FIFO, OS priority 47
Java priority 23: SCHED_FIFO, OS priority 49
Java priority 24: SCHED_FIFO, OS priority 51
Java priority 25: SCHED_FIFO, OS priority 53
Java priority 26: SCHED_FIFO, OS priority 55
Java priority 27: SCHED_FIFO, OS priority 57
Java priority 28: SCHED_FIFO, OS priority 59
Java priority 29: SCHED_FIFO, OS priority 61
Java priority 30: SCHED_FIFO, OS priority 63
Java priority 31: SCHED_FIFO, OS priority 65
Java priority 32: SCHED_FIFO, OS priority 67
Java priority 33: SCHED_FIFO, OS priority 69
Java priority 34: SCHED_FIFO, OS priority 71
Java priority 35: SCHED_FIFO, OS priority 73
Java priority 36: SCHED_FIFO, OS priority 75
Java priority 37: SCHED_FIFO, OS priority 77
Java priority 38: SCHED_FIFO, OS priority 79
```

Internal base priorities:

```
Internal Java priority 39: SCHED_FIFO, OS priority 81
Internal Java priority 40: SCHED_FIFO, OS priority 83
Internal Java priority 41: SCHED_FIFO, OS priority 84
Internal Java priority 42: SCHED_FIFO, OS priority 85
Internal Java priority 43: SCHED_FIFO, OS priority 86
Internal Java priority 44: SCHED_FIFO, OS priority 87
Internal Java priority 45: SCHED_FIFO, OS priority 88
OS priorities 11, 12, 13
OS priorities even numbers 26, 28, 30, ..., 82
OS priority 89
```

See also: the "Synchronization" section in http://www.rtsj.org/specjavadoc/book_index.html.

Related concepts

"Introduction to the Metronome Garbage Collector" on page 49
The benefit of the Metronome Garbage Collector is that the time it takes is more predictable and garbage collection can take place at set intervals over a period of time.

"Launching secondary processes" on page 20
The java.lang.Runtime.exec methods in the Java virtual machine (JVM) API give your Java application the ability to execute a command in a separate process.

Priority inheritance

The active priority of a thread can be temporarily boosted because it holds a lock required by a higher priority thread. These locks might be internal JVM locks or user-level monitors associated with synchronized methods or synchronized blocks. The priority of a regular Java thread, therefore, might temporarily have a real-time priority until the point at which the thread has released the lock.

One consequence of priority inheritance means that the thread policy of a SCHED_OTHER thread is temporarily changed to SCHED_FIFO.

For more information about base and active priorities, see the "Synchronization" section in the RTSJ specification.

Memory management

Garbage collecting memory heaps has always been considered an obstacle to real-time programming because of the unpredictable behavior introduced by garbage collection. The Metronome garbage collector in IBM WebSphere Real Time for RT Linux can provide high deterministic GC performance. At the same time, the Real-Time Specification for Java (RTSJ) provides several extensions to the memory model for objects outside the garbage-collected heap, so that a Java programmer can explicitly manage both short-lived and long-lived objects.

Memory areas

The RTSJ introduces the concept of a memory area that can be used for the allocation of objects. Some memory areas exist outside the heap and place restrictions on what the system and garbage collector can do with objects. For example, objects in some memory areas are never garbage collected but the Garbage Collector can scan these memory areas for references to any object in the heap to preserve the integrity of the heap.

Memory management has three basic types:

- Heap memory is the traditional Java heap but is managed by the Metronome Garbage Collector.
- Scoped memory must be specifically requested by applications and can be used only by real-time threads, including no-heap real-time threads and no-heap asynchronous event handlers.
- Immortal memory represents an area of memory containing objects that can be referenced by any schedulable object, specifically including no-heap real-time threads and no-heap asynchronous event handlers. It is used by class loading and static initialization even if the application does not use it.

Immortal or scoped memory can be designated to use physical memory, which consists of memory regions having specific characteristics such as substantially

faster access. In general, physical memory is not used often and is unlikely to affect the standard JVM user.

Heap memory

The maximum size is controlled by **-Xmx** but remember *not* to set the initial heap size (**-Xms**) or set it equal to maximum heap size **-Xmx**, because, in real time, the heap never expands from the initial heap size to the maximum heap size. When you reach maximum heap size with no free space, OutOf MemoryError results. In general, the real time JVM consumes more heap memory than the traditional JVM because supporting deterministic collection requires objects to be organized differently, resulting in higher heap fragmentation. In addition, arrays are broken into fragments, each of which has a header. It depends on the ratio of large to small objects and the amount of array usage, but it is likely to find an application needing 20% more heap space.

The Metronome Garbage Collector is similar to the "mostly concurrent" collector that exists in the mainstream JVM in that it collects garbage while the application is running. In a perfect world, the collection cycle completes before the application runs out of memory, but some applications with very high allocation rates can allocate faster than the Metronome Garbage Collector can collect. Various detailed controls affect the collection rate, but there is one control that forces Metronome to revert to traditional stop-the-world GC before finally throwing OutOf MemoryError. The runtime parameter is **-Xgc:synchronousGCOnOOM** and the counterpart is **-Xgc:nosynchronousGCOnOOM**. The default is **-Xgc:synchronousGCOnOOM**.

Scoped memory

The RTSJ introduces the concept of scoped memory. It can be used by objects that have a well-defined lifetime. A scope can be entered explicitly, or it can be attached to a schedulable object (a real-time thread or an asynchronous event handler) that effectively enters the scope before it runs the run() method of the object. Each scope has a reference count and when this reaches zero the objects that are resident in that scope can be closed (finalized) and the memory associated with that scope is released. Reuse of the scope is blocked until finalization is complete.

Scoped memory can be divided into two types: VTMemory and LTMemory. These types of scoped memory vary by the time required to allocate objects from the area. LTMemory guarantees linear time allocation when memory consumption from the memory area is less then the initial size of the memory area. VTMemory offers no such guarantee.

Scopes can be nested. When a nested scope is entered, all subsequent allocations are taken from the memory associated with the new scope. When the nested scope is exited, the previous scope is restored and subsequent allocations are again taken from that scope.

Because of the lifetimes of scoped objects, it is necessary to limit the references to scoped objects, by means of a restricted set of assignment rules. A reference to a scoped object cannot be assigned to a variable from an outer scope, or to a field of an object in either the heap or the immortal area. A reference to a scoped object can be assigned only into the same scope or into an inner scope. The virtual machine detects incorrect assignment attempts and throws an IllegalAssignmentError exception when they occur. The flexibility provided in

choice of scoped memory types allows the application to use a memory area that has characteristics that are appropriate to a particular syntactically defined region of the code.

The size of the area must be specified during construction of the area and a command-line parameter, **-Xgc:scopedMemoryMaximumSize**, controls the maximum value. The default is 8 MB and is adequate for most purposes.

Immortal memory

Immortal memory is a memory resource shared among all schedulable objects and threads in an application. Objects allocated in immortal memory are always available to non-heap threads and asynchronous event handlers and are not subject to delays caused by garbage collection. Objects are freed by the system when the program terminates.

The size is controlled by **-Xgc:immortalMemorySize**; for example, **-Xgc:immortalMemorySize=20m** sets 20 MB, The default is 16 MB, which is usually adequate, unless you are doing a lot of class loading. Class loading is the likely cause of most OutOfMemoryError exceptions.

Related concepts

"Introduction to the Metronome Garbage Collector" on page 49
The benefit of the Metronome Garbage Collector is that the time it takes is more predictable and garbage collection can take place at set intervals over a period of time.

Estimating memory requirements

How to obtain information required to allocate sufficient memory

A reasonable approach is to identify the memory required to hold the expected objects, with a sensible safety margin. Analysis of the application helps identify the number and nature of objects required, although the actual size required for an object can vary between different systems. Using the SizeEstimator class takes actual object size into account, providing more portable information.

The SizeEstimator class

The SizeEstimator class provides guidance information about the amount of memory needed to store an object. The estimate is an indication of the minimum memory space that should be allocated for the object itself, and does not take into account memory requirements for any other resources that might be required by the object, for example during its construction.

For details about this class, see http://www.rtsj.org/specjavadoc/book_index.html

Using memory

A comparison of Java threads, real-time threads, and no-heap real-time threads.

The Real-Time Specification for Java (RTSJ) adds two classes to support real-time threads: RealtimeThread class and NoHeapRealtimeThread class.

- Both real-time threads and no-heap real-time threads are schedulable objects. As schedulable objects, they have the following parameters: release, scheduling, memory, and processing group.

- Real-time threads can access objects in heap memory as well as in scoped and immortal memory.
- No-heap real-time threads access only scoped and immortal memory areas.
- No-heap real-time threads need a higher priority than other real-time threads. If their priority is less than other real-time threads, they lose their advantage of running without interference from the Garbage Collector.

Note: A no-heap real-time thread with priority higher than other real-time threads will not be interrupted by garbage collection.

Table 10. Memory access by real-time and no-heap real-time threads

Threads	Immortal memory	Scoped memory	Heap memory
Normal threads	✓	✗	✓
Real-time threads	✓	✓	✓
no-heap real-time threads	✓	✓	✗

Types of memory area

Immortal memory

Immortal memory is not subject to garbage collection. After space has been allocated in immortal memory, the space cannot be reclaimed until the application exits.

- Because of the nature of immortal memory, you might want to find ways to reuse the memory. One possibility is to create a pool of reusable objects. Using scoped memory is an alternative.
- Objects in immortal memory cannot reference anything in scoped memory. If a field of an object in immortal memory is assigned an object from scoped memory, an IllegalAssignmentError exception is thrown.

Scoped memory

Scoped memory can be used as the initial memory area of a schedulable object or can be entered by one. When no longer referenced, the area is cleared of all objects. Schedulable objects running in a scoped memory area perform all their object allocations from that area. When a scoped memory area is unused, the objects inside it are finalized and the memory is reclaimed, preparing the scope for reuse. When the scoped memory area is no longer available to any schedulable objects, the memory is reclaimed for other uses.

The memory area described by a ScopedMemory instance does not exist in the Java heap and is not subject to garbage collection. It is safe to use a ScopedMemory object as the initial memory area associated with a NoHeapRealtimeThread or to enter the memory area using the ScopedMemory.enter method inside a NoHeapRealtimeThread.

Physical memory

Use physical memory when the characteristics of the memory itself are important; for example, non-pageable or non-volatile.

Linear time allocation scheme (LTMemory)
LTMemory represents a memory area guaranteed by the system to have linear time allocation when memory consumption from the memory area is less than the initial size of the memory area. Run time for allocation is allowed to vary when memory consumption is between the initial size and the maximum size for the area. Furthermore, the underlying system is not required to guarantee that memory between initial and maximum is always available.

Variable time allocation scheme (VTMemory)
VTMemory is similar to LTMemory except that the running time of an allocation from a **VTMemory** area need not complete in linear time.

Heap Memory
Objects in heap memory cannot reference anything in scoped memory. If a field of an object in heap memory is assigned an object from scoped memory, an IllegalAssignmentError exception is thrown.

Synchronization and resource sharing

In a real-time system, when three or more threads that are running at different priorities and are synchronizing with each other, a condition called priority inversion can occasionally result, in which a higher priority thread is blocked from running by a lower priority thread for an extended period of time. WebSphere Real Time for RT Linux uses a scheme called priority inheritance to avoid this condition.

When a higher priority task is blocked from running by a lower priority task, the priority of the lower priority task is temporarily boosted to match the higher priority until the higher priority task is no longer blocked.

Periodic and aperiodic parameters

Real-time threads have a number of release parameters, determining how often a schedulable object is released. Periodic and aperiodic parameters are examples of release parameters.

Periodic parameters

This class is for those schedulable objects that are released at regular intervals.

AbsoluteTime
Is expressed in milliseconds and nanoseconds.

RelativeTime
Is the length of time of a given event expressed in milliseconds and nanoseconds. For example, you can measure the absolute time when an event starts and finishes. You can then calculate the relative time as the difference between the two measurements.

Aperiodic parameters

This class is used by those schedulable objects that are released at irregular intervals. Because a second aperiodic event might occur before the first one has completed, you can define the length of the queue of outstanding requests.

Asynchronous event handling

Asynchronous event handlers react to events that occur outside a thread; for example, input from an interface of an application. In real-time systems, these events must respond within the deadlines that you set for your application.

Asynchronous events can be associated with system interrupts and POSIX signals, and asynchronous events can be linked to a timer.

Like real-time threads, asynchronous event handlers have a number of parameters associated with them. For a list of these parameters, see "Schedulables and their Parameters" on page 61.

Signal Handlers

The POSIXSignalHandler supports the signals SIGQUIT, SIGTERM, and SIGABRT. The default behavior for SIGQUIT causes a Javadump to be generated. The generation of the Javadump does not interfere with the operation of a running program, apart from CPU time and file reading and writing. The generation of a Javadump interrupts the program until the Javadump has been completed; application performance will not be predictable while Javadumps are being generated.

To suppress all core and Javadump generation on a failure, use **-Xdump:none**.

To suppress only system dump and Javadump generation on a SIGQUIT signal, specify **-Xdump:java:none -Xdump:java:events=gpf+abort**.

The following signals can be attached to asynchronous event handlers (AEHs) by the POSIXSignalHandler mechanism (signal descriptions as defined in `/usr/include/bits/signum.h`):

```
#define SIGQUIT         3       /* Quit (POSIX).  */
#define SIGABRT         6       /* Abort (ANSI).  */
#define SIGKILL         9       /* Kill, unblockable (POSIX).  */
```

No other signals are currently supported. All of the above are asynchronous signals, and it is impossible to support attaching to synchronous signals (such as SIGILL and SIGSEGV) because they indicate a failure of your application or the JVM code, not an externally generated event.

Note: SIGQUIT by default causes the Java application to generate dumps (for example, a Javadump) when received by the JVM. Although additionally it is delivered to any attached AEH, this delivery might cause confusing or undesirable behavior and you can disable it by using the **-Xdump:none:events=user** option on the Java command line.

Required documentation

WebSphere Real Time for RT Linux implements the Real-Time Specification for Java (RTSJ).

WebSphere Real Time for RT Linux version 2.0 has been certified as RTSJ 1.0.2 compliant against the RTSJ Technology Compatibility Kit version 3.0.13 FCS and is compliant with the Java Compatibility Kit (JCK) for version 6.0.

Supported facilities

The following facilities are supported:

- Allocation-rate enforcement on heap allocation to limit the rate at which a schedulable object creates objects in the heap.

Unsupported facilities

The following facilities are not supported:

- Priority Ceiling Emulation Protocol. For example, it does not permit PriorityCeilingEmulation to be used as a monitor control policy.
- Atomic access support, except where required for conformance to the specification.
- No schedulers other than the base priority scheduler are available to applications.
- Cost enforcement.

Required documentation for Real-Time Specification for Java

The *Required Documentation* section of the Real-Time Specification for Java (RTSJ) is quoted in this section. Any deviations from the standard implementation of RTSJ are noted.

1. **The feasibility testing algorithm is the default.**

   "If the feasibility testing algorithm is not the default, document the feasibility testing algorithm."
2. **Only the base priority scheduler is available to applications.**

   "If schedulers other than the base priority scheduler are available to applications, document the behavior of the scheduler and its interaction with each other scheduler as detailed in the Scheduling chapter. Also document the list of classes that constitute schedulable objects for the scheduler unless that list is the same as the list of schedulable objects for the base scheduler."
3. **A schedulable object that is preempted by a higher priority schedulable object will be placed at the front of the queue for its priority.**

   "A schedulable object that is preempted by a higher-priority schedulable object is placed in the queue for its active priority, at a position determined by the implementation. If the preempted schedulable object is not placed at the front of the appropriate queue the implementation must document the algorithm used for such placement. Placement at the front of the queue can be required in a future version of this specification."
4. **Cost enforcement is not supported.**

   "If the implementation supports cost enforcement, the implementation is required to document the granularity at which the current CPU consumption is updated."
5. **Simple sequential mapping is supported.**

   "The memory mapping implemented by any physical memory type filter must be documented unless it is a simple sequential mapping of contiguous bytes."
6. **There are no subclasses for the Metronome Garbage Collector supplied with WebSphere Real Time for RT Linux.**

   "The implementation must fully document the behavior of any subclasses of GarbageCollector."

7. **There are no MonitorControl subclasses supplied with WebSphere Real Time for RT Linux.**

   "An implementation that provides any MonitorControl subclasses not detailed in this specification must document their effects, particularly with respect to priority inversion control and which (if any) schedulers fail to support the new policy."

8. **A schedulable object holding a monitor required by a higher priority schedulable object has its priority boosted to the higher priority until such time as it releases the monitor. If, at that point, the schedulable object is to be made no longer runnable (that is, there is higher priority work to be done) it will be placed at the back of the queue for its original (unboosted) priority when running on kernels prior to SUSE Linux Enterprise Real Time 10 SP2 update kernel version 2.6.22.19-0.16, and Red Hat Enterprise Linux 5.1 MRG 2.6.24.7-73 Errata 1. Kernels at these levels or later place the schedulable object at the front of the queue.**

   "If on losing "boosted" priority because of a priority inversion avoidance algorithm, the schedulable object is not placed at the front of its new queue, the implementation must document the queuing behavior."

9. **The base scheduler is the only scheduler provided with WebSphere Real Time for RT Linux.**

   "For any available scheduler other than the base scheduler an implementation must document how, if at all, the semantics of synchronization differ from the rules defined for the default PriorityInheritance instance. It must supply documentation for the behavior of the new scheduler with priority inheritance (and, if it is supported, priority ceiling emulation protocol) equivalent to the semantics for the base priority scheduler found in the Synchronization chapter."

10. **The worst case time from the firing of an event to the scheduling of an associated bound event handler will average 40µs and not exceed 100µs providing that there are no competing schedulable objects or system activity of equal or higher priority and providing that garbage collection does not interfere. If the schedulable object driving the fire method, the AsyncEvent object or the handler references the heap then the potential influence of garbage collection is as documented in (A). This assumes that the code is being interpreted and that a single handler (which is bound) is configured on the event.**

    "The worst-case response interval between firing an AsyncEvent because of a bound happening to releasing an associated AsyncEventHandler (assuming no higher-priority schedulable objects are runnable) must be documented for some reference architecture."

11. **The worst case interval between firing an AsynchronouslyInterruptedException at an ATC-enabled thread and the first delivery of that exception will average 35µs and not exceed 160µs providing that there are no competing schedulable objects or system activity of equal or higher priority and providing that garbage collection does not interfere. ATC-enabled in this case means that the thread is executing in an AI enabled method in a region that is not ATC-deferred and those conditions remain true until delivery of the exception. The potential influence of garbage collection is as documented in (A). If the target thread is in native code then the delay is potentially unbounded. This assumes that the code is being interpreted.**

"The interval between firing an AsynchronouslyInterruptedException at an ATC-enabled thread and first delivery of that exception (assuming no higher-priority schedulable objects are runnable) must be documented for some reference architecture."

12. **Not applicable see response 4.**

    "If cost enforcement is supported, and the implementation assigns the cost of running finalizers for objects in scoped memory to any schedulable object other than the one that caused the scope's reference count to drop to zero by leaving the scope, the rules for assigning the cost shall be documented."

13. **There are no changes to the standard implementation of RealtimeSecurity.**

    "If the implementation of RealtimeSecurity is more restrictive than the required implementation, or has runtime configuration options, those features shall be documented."

14. **The finalizers for objects in a scoped memory area will be run by the last thread to reference that area, that is, they will be run when the thread decrements the reference count from one to zero. Any cost associated with running the finalizers will be assigned to that thread.**

    "An implementation can run finalizers for objects in scoped memory before the scope is reentered and before it returns from any call to getReferenceCount() for that scope. It must, however, document when it runs those finalizers."

15. **The resolution is not settable.**

    "For each supported clock, the documentation must specify whether the resolution is settable, and if it is settable the documentation must indicate the supported values."

16. **There are no other clocks other than the real-time clock provided with WebSphere Real Time for RT Linux.**

    "If an implementation includes any clocks other than the required real-time clock, their documentation must indicate in what contexts those clocks can be used."

Note:

A The reference architecture for the tests will be an LS20, 4–way, 2 GHz with 1 MB cache and 4 GB of memory.

B Garbage collection can cause a delay at any point in a thread that is associated with the heap. The collector can operate in one of two basic modes governing the behavior when heap memory is exhausted. If the collector is set to immediately throw OutOfMemoryError in these circumstances, the worst case garbage collection delay will typically be below 1 ms. Currently, in some circumstances, the delay can be higher; for example, if there are many threads with deeply nested stacks or large numbers of large-sized scopes. If the collector is set to perform a synchronous GC before throwing an OutOfMemoryError, the potential collection delay is related to the number of live objects in the heap and the numbers of objects in other memory areas. In these circumstances, the delay is considered to be unbounded because it can be many seconds for typical heap sizes.

Chapter 8. The sample application

The sample application uses a series of examples to demonstrate the features of WebSphere Real Time for RT Linux that can be used to improve the real-time characteristics of Java programs.

The source files for the sample application are in the `demo/realtime/sample_application.zip` file.

The sample consists of two main components:

- **A simulation,** a simple example of a lunar lander. Its position is defined by its height above the ground and the distance from the landing area. See Figure 3 on page 76.

  The simulation class is written using no-heap real-time threads (NHRTs) and is not modified any further in this documentation.
- **A controller** that sends commands to the simulation. It sends radar pings to judge the lander's height and control the rate of descent of the lander based on this information. The controller also receives a stream of information from the lander; for example, the lander's distance from the landing area.

  The controller is written initially in standard Java. In "Modifying Java applications" on page 90, it is developed into a real-time Java program

Depending on the outcome of the landing, the controller is sent one of two messages: either crash or land.

Using the sample application you can perform these operations:

- Run both the simulation and the controller together to demonstrate a combination of real-time and standard Java classes running together. For information, see "Building the sample application" on page 76 and "Running the sample application" on page 77, where you will also see output that you can expect from the sample application.

  Note: You can start both the simulation and the controller at the same time using the LaunchBoth class.
- Compare the difference when using the Metronome Garbage Collector and the standard Garbage Collector. For information, see "Running the sample application without Real Time" on page 77 and "Running the sample application with Metronome Garbage Collector" on page 79.
- Run the application using the ahead-of-time (AOT) complier. For information, see "Running the sample application using AOT" on page 80.

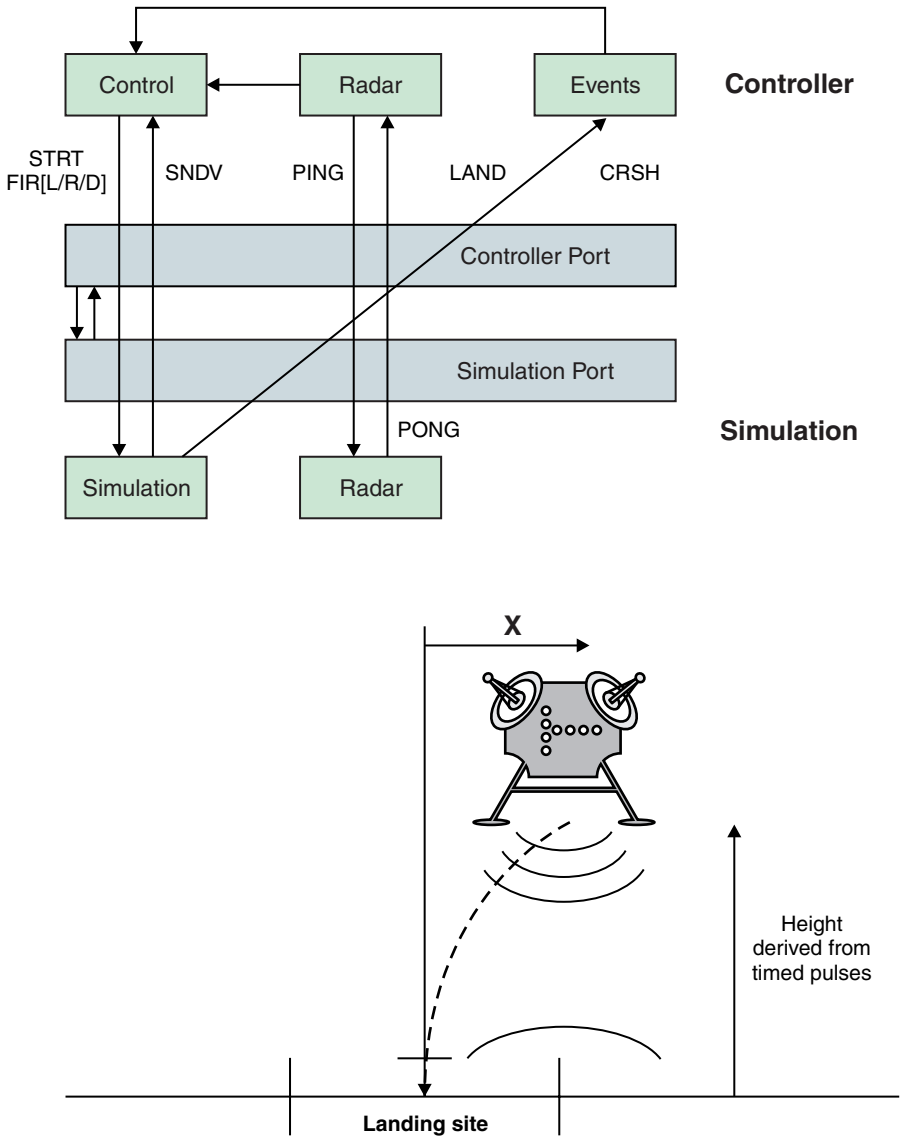


Figure 3. Diagram of the lunar lander

This diagram shows the relationship of the modules provided in the sample. The upper part of the diagram shows the controller and simulator. The controller has three threads: control, radar, and events threads. The simulator has two threads: the simulator and the radar thread. The lower part of the diagram shows the lunar lander and indicates the two control functions of left and right together with the pulses that determine the height of the lander.

Building the sample application

The sample application source code is provided for guidance. Preparation requires unpacking and compilation of the Java source code before it can be run.

Procedure

1. Create a working directory.
2. Extract the sample application into your working directory:

```
unzip sample_application.zip
```

3. Create a new directory for your output:
   ```
   mkdir classes
   ```
4. Compile the source.
   a. Generate a list of the files:
      ```
      find -name "*.java" > source
      ```
   b. Compile the source:
      ```
      javac -Xrealtime -Xlint:deprecated -g -d classes @source
      ```
   c. Create a jar file of the class files:
      ```
      jar cf demo.jar -C classes/ .
      ```

What to do next

You can now run the sample application.

Running the sample application

WebSphere Real Time provides a standard JVM as well as a real-time JVM, started with the **-Xrealtime** command-line argument.

The sample application has two components, designed to be run in separate JVMs:

- The Simulation, which only runs in Real-Time Java.
- The Controller, which can run either non-Real-Time or Real-Time Java.

Running the Controller code in a variety of modes demonstrates the benefits of the IBM Real-Time Java technology.

Running the sample application without Real Time

In this procedure, you run the sample application without taking advantage of IBM WebSphere Real Time.

Before you begin

To run the sample application, you must first build the sample source code. See "Building the sample application" on page 76 for more information.

Procedure

1. Start the simulation:
   ```
   java -Xrealtime -classpath ./demo.jar -Xgc:scopedMemoryMaximumSize=11m demo.sim.SimLauncher <p
   ```
   where *<port>* is an deallocated port for this workstation.
2. Start the controller:
   ```
   java -classpath ./demo.jar -mx300m demo.controller.JavaControlLauncher <host> <port>
   ```
   where *<host>* is the hostname of the workstation running the simulation and *<port>* is the port specified in the previous step.

Results

The output of this application produces the following message showing that the simulation and the controller have started:

```
SimLauncher: Waiting for connections...
Starting control thread...
```

Some point samples of the values in the controller are printed out to the console:

```
x=99.50, radar=199.11, y=198.34, vx=-0.71, vy=-0.43, timeSinceLast=0.19, targetVx=-6.01, targetVy=-9.00
x=95.50, radar=194.59, y=192.70, vx=-2.70, vy=-2.43, timeSinceLast=0.20, targetVx=-5.94, targetVy=-9.00
x=87.50, radar=186.57, y=183.06, vx=-4.70, vy=-4.40, timeSinceLast=0.20, targetVx=-5.77, targetVy=-9.00
x=76.46, radar=172.84, y=169.42, vx=-5.42, vy=-6.75, timeSinceLast=0.20, targetVx=-5.60, targetVy=-9.00
x=65.36, radar=155.58, y=151.84, vx=-5.50, vy=-9.19, timeSinceLast=0.20, targetVx=-5.57, targetVy=-9.00
x=54.36, radar=138.06, y=135.24, vx=-5.44, vy=-7.63, timeSinceLast=0.20, targetVx=-5.56, targetVy=-9.00
x=43.26, radar=120.57, y=117.22, vx=-5.67, vy=-9.62, timeSinceLast=0.20, targetVx=-5.52, targetVy=-9.00
x=32.36, radar=103.60, y=100.72, vx=-5.47, vy=-9.06, timeSinceLast=0.20, targetVx=-5.43, targetVy=-9.00
x=21.52, radar=84.60, y=82.86, vx=-5.32, vy=-9.09, timeSinceLast=0.20, targetVx=-5.60, targetVy=-9.00
x=10.72, radar=67.07, y=65.56, vx=-5.30, vy=-10.54, timeSinceLast=0.20, targetVx=-5.65, targetVy=-9.00
x=0.76, radar=51.08, y=49.78, vx=-4.30, vy=-7.52, timeSinceLast=0.20, targetVx=-0.50, targetVy=-9.00
x=-5.24, radar=37.07, y=35.94, vx=-2.30, vy=-8.26, timeSinceLast=0.20, targetVx=0.50, targetVy=-9.00
x=-7.24, radar=20.05, y=19.90, vx=-0.30, vy=-6.15, timeSinceLast=0.20, targetVx=0.50, targetVy=-9.00
x=-6.36, radar=2.68, y=2.80, vx=0.27, vy=-10.08, timeSinceLast=0.20, targetVx=0.50, targetVy=-9.00
```

Just before the simulation stops, the following message is issued:

```
Fire down transitions 141, fire horizontally transitions 141
LAND!
```

In addition to this, the controller produces a graph called `graph.svg` in the same directory. Figure 4 on page 79 shows the effect of garbage collection pauses on the JavaRadar thread with a standard non-Real-Time JVM. The spikes in the blue Radar Height line show how standard garbage collection pauses affect the Controller application. On some runs, the garbage collection pauses are long enough to cause failures, leading to the message:

```
CRASH!
```

To see the garbage collection pause times, add the **-verbose:gc** option to the controller launch command:

```
java -classpath ./demo.jar -verbose:gc -mx300m demo.controller.JavaControlLauncher <host> <port>
```

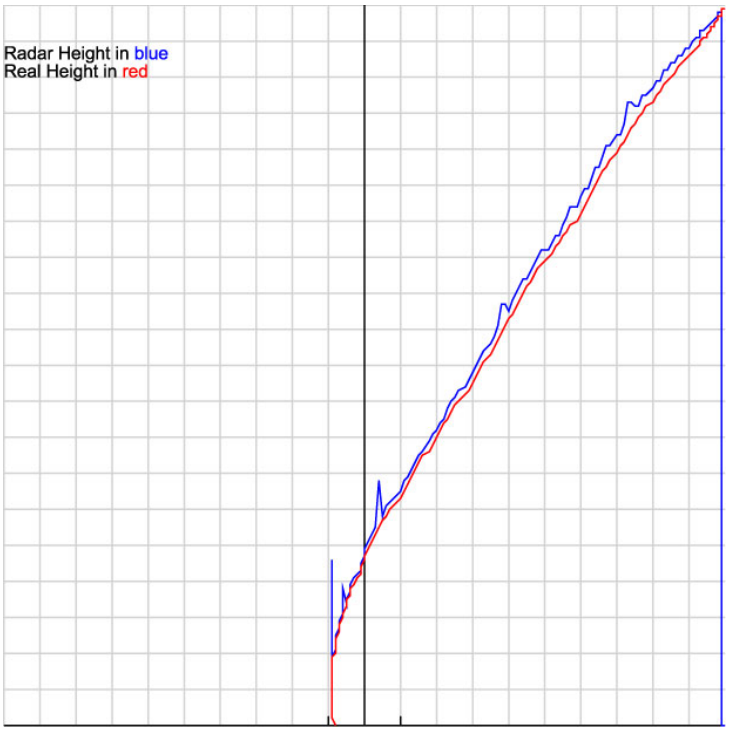


Figure 4. Graph generated by the sample application without Real Time

Running the sample application with Metronome Garbage Collector

You can run a standard Java application in a real-time environment without any need to rewrite the code, by adding the -Xrealtime option, which enables both Real-Time Java language features and the Metronome Garbage Collector.

Before you begin

To run the sample application, you must first build the sample source code. See "Building the sample application" on page 76 for more information.

Procedure

1. Start the simulation:

   ```
   java -Xrealtime -classpath ./demo.jar -Xgc:scopedMemoryMaximumSize=11m demo.sim.SimLauncher <p
   ```

   where *<port>* is an deallocated port for this workstation.
2. Start the controller:

   ```
   java -Xrealtime -classpath ./demo.jar -mx300m demo.controller.JavaControlLauncher <host> <port
   ```

   where *<host>* is the hostname of the workstation running the simulation and *<port>* is the port specified in the previous step. Running both JVMs on the same workstation can lead to less deterministic behavior. See "Considerations" on page 5 for more information.

Results

As with the non-Real-Time example, the application will now run and generate data points and a graph.

In this run with Metronome garbage collection, the graph shows no spikes in the blue Radar Height, and very accurate tracking of the red Real Height line, because the Controller code is now running with only very short garbage collection pauses.

The difference between the short, multiple Metronome garbage collection pauses (typically less than 1 millisecond) and the longer, fewer pauses of non-real-time garbage collection (typically 10s or 100s of milliseconds) can be seen by adding the **-verbose:gc** option to the Controller run command.

See "Using verbose:gc information" on page 51 for more information on the verbose garbage collection output.

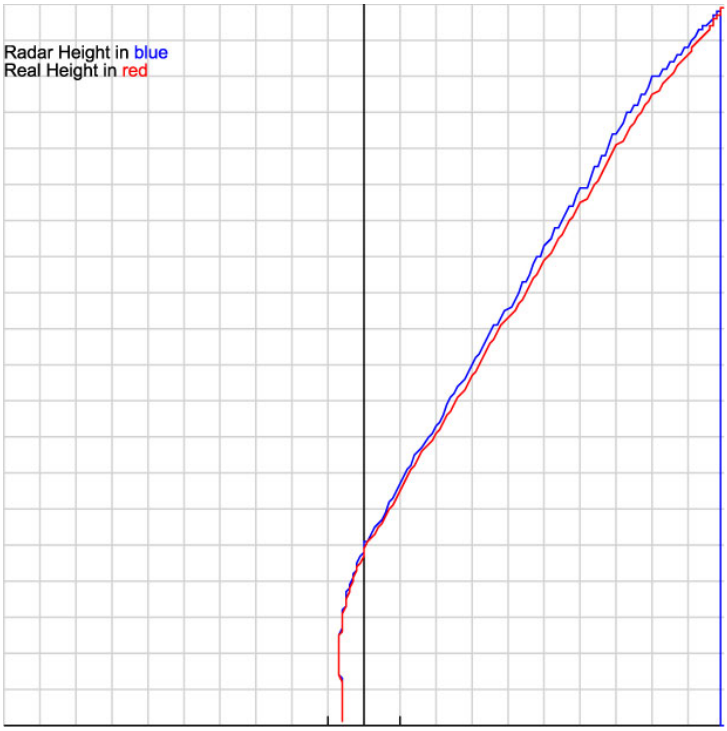


Figure 5. Graph generated by the sample application with Real Time

Running the sample application using AOT

This procedure runs a standard Java application in a real-time environment using the ahead-of-time (AOT) compiler, without the need to rewrite code. Use this sample to compare running the same application using the JIT compiler.

See Chapter 4, "Using compiled code with WebSphere Real Time for RT Linux," on page 23 for more details about ahead-of-time compilation.

Before you begin

To run the sample application, you must first build the sample source code. See "Building the sample application" on page 76 for more information.

About this task

The ahead-of-time compiler compiles your Java application to native code before running it. Therefore, you can predict how the application runs more precisely, because there are no interruptions caused by just-in-time (JIT) compilation.

Procedure

1. Convert the application bytecodes into native code.
   a. Identify which bytecodes to precompile by running the sample with the normal JIT compiler:

      ```
      java -Xrealtime -Xjit:verbose={precompile},vlog=./sim.aotOpts \
           -classpath ./demo.jar -Xgc:scopedMemoryMaximumSize=11m demo.sim.SimLauncher <port>
      ```

      And in a new window, enter:

      ```
      java -Xrealtime -Xjit:verbose={precompile},vlog=./control.aotOpts \
           -classpath ./demo.jar -Xmx300m demo.controller.JavaControlLauncher localhost <port>
      ```

      where <*port*> is an deallocated port for this workstation. You will see messages similar to the following:

      ```
      Fire down transitions 141, fire horizontally transitions 141
      ```

      and:

      ```
      Land!
      ```

   b. Combine the AOT options files. The names used for the log files created in the previous step have date and process ID information appended to the file name specified by the **vlog=** option. For example, **vlog=sim.aotOpts** generates a file name similar to **sim.aotOpts.20081014.234958.13205**:

      ```
      cat sim.aotOpts.20081014.234958.13205 control.aotOpts.20081014.234958.13205 > sample.aotOpt
      ```

   c. Compile the files in the `sample.aotOpts` file in `realtime.jar`, `vm.jar`, `rt.jar`, and the application `demo.jar`. When using shared class caches, the name of the cache must not exceed 53 characters.

      ```
      admincache -Xrealtime -populate -cacheName "sample" -aotFilterFile sample.aotOpts -classpat
      $JAVA_HOME/jre/lib/i386/realtime/jclSC160/vm.jar \
      $JAVA_HOME/jre/lib/i386/realtime/jclSC160/realtime.jar \
      $JAVA_HOME/jre/lib/rt.jar \
      ./demo.jar
      ```

      You see output similar to the following:

      ```
      J9 Java(TM) admincache 1.0
      Licensed Materials - Property of IBM

      (c) Copyright IBM Corp. 1991, 2008 All Rights Reserved
      IBM is a registered trademark of IBM Corp.
      Java and all Java-based marks and logos are trademarks or registered
      trademarks of Sun Microsystems, Inc.

      JVMSHRC256I Persistent shared cache "sample" has been destroyed
      Converting files
      Converting /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/vm.jar into shared class
      Succeeded to convert jar file /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/vm.jar
      Converting /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/realtime.jar into shared
      Succeeded to convert jar file /team/mstoodle/demo/sdk/jre/lib/i386/realtime/jclSC160/realti
      Converting /team/mstoodle/demo/sdk/jre/lib/rt.jar into shared class cache
      Succeeded to convert jar file /team/mstoodle/demo/sdk/jre/lib/rt.jar
      Converting /team/mstoodle/demo/demo.jar into shared class cache
      Succeeded to convert jar file /team/mstoodle/demo/demo.jar

      Processing complete
      ```

Note: The line:

```
JVMSHRC256I Persistent shared cache "sample" has been destroyed
```

means that any existing cache called "sample" will be destroyed by this command, to create the specified cache.

d. Display the contents of the populated cache:

```
admincache -Xrealtime -cacheName "sample" -printStats
```

2. Start the simulation:

```
java -Xrealtime -Xnojit -Xmx300m -Xshareclasses:name="sample" \
     -classpath ./demo.jar -Xgc:scopedMemoryMaximumSize=11m \
     demo.sim.SimLauncher <port>
```

where <*port*> is an unallocated port for this workstation.

3. Start the controller:

```
java -Xrealtime -Xnojit -Xmx300m -Xshareclasses:name="sample" \
     -classpath ./demo.jar \
     demo.controller.JavaControlLauncher <host> <port>
```

where <*host*> is the hostname of the workstation running the simulation and <*port*> is the port specified in the previous step. Running both JVMs on the same workstation can lead to less deterministic behavior. See “Considerations” on page 5 for more information.

Results

As with the non-real-time example, the application now runs and generates data points and a graph.

In this run with ahead-of-time compilation, the graph shows no spikes in the blue Radar Height, and very accurate tracking of the red Real Height line, because the Controller code is now running with only very short garbage collection pauses and no just-in-time compilation interruptions.

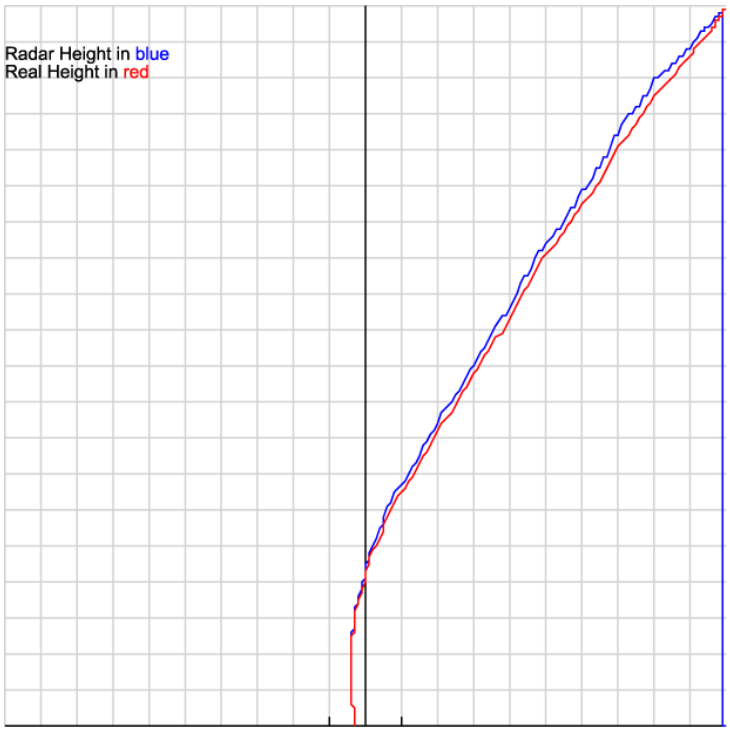


Figure 6. Graph generated by the sample application with ahead-of-time compilation

A benefit of using the shared class cache to run this application is that the controller and the simulation JVMs share some of the memory used by classes loaded by both JVMs.

Related concepts

"The ahead-of-time compiler" on page 25
Ahead-of-Time (AOT) compilation allows you to compile Java classes and methods before you run your code. AOT compilation avoids the unpredictable timing effect the JIT compiler can have on sensitive performance paths. To ensure your code is compiled before it executes and for the highest level of deterministic performance, you can precompile your code into a shared class cache using the AOT compiler.

Related tasks

"Using the AOT compiler" on page 27
Use these steps to precompile your Java code. This procedure describes the use of the **-Xrealtime** option in a javac command, the admincache tool, and the **-Xrealtime** and **-Xnojit** options with the java command.

Related reference

"Ahead-of-time options" on page 329
The definitions for the ahead-of-time options.

Chapter 9. The sample real-time hash map

The sample application uses a series of examples to demonstrate the features of WebSphere Real Time for RT Linux that can be used to improve the real-time characteristics of Java programs.

The standard java.util.HashMap that IBM provides works well for high throughput applications. It also helps with applications that know the maximum size their hash map needs to grow to. For applications that need a hash map that could grow to variable sizes, depending on usage, there is a potential performance problem with the standard hash map. The standard hash map provides good response times for adding new entries into the hash map using the put method. However, when the hash map fills up, a larger backing store must be allocated. This means that the entries in the current backing store must be migrated. If the hash map is large, the time to perform a put could also be large. For example, the operation could take several milliseconds.

WebSphere Real Time for RT Linux includes a sample real-time hash map. It provides the same functional interface as the standard java.util.HashMap, but enables much more consistent performance for the put method. Instead of creating a backing store and migrating all the entries when the hash map fills up, the sample hash map creates an additional backing store. The new backing store is chained to the other backing stores in the hash map. The chaining initially causes a slight performance reduction while the empty backing store is allocated and chained to the other backing stores. Once the backing hash map is updated, it is faster than having to migrate all the entries. A disadvantage of the real-time hash map is that the get, put and remove operations are slightly slower. The operations are slower because each look-up must to proceed through a set of backing hash maps instead of just one.

To try out the real-time hash map, add the `RTHashMap.jar` file to the start of your boot class path. If you installed WebSphere Real Time for RT Linux into the directory `$WRT_ROOT`, then add the following option to use the real-time hash map with your application, instead of the standard hash map:

```
-Xbootclasspath/p:$WRT_ROOT/demo/realtime/RTHashMap.jar
```

The source and class files for the real-time hash map implementation are included in the `demo/realtime/RTHashMap.jar` file. In addition, a real time java.util.LinkedHashMap and java.util.HashSet implementation are also provided.

Chapter 10. Writing Java applications to exploit real time

These examples describe how to exploit the real-time environment. They range from the simplest example, running a Java application in real time without any modifications to the code, through to a more complex process of planning and writing no-heap real-time threads. Reasons are provided to help you decide which approach might be most suitable for your applications.

Introduction to writing real-time applications

You do not have to write elaborate no-heap real-time applications to exploit the features of real-time technology. Some of the benefits can be used with very little change to your existing code.

For application programers, here are the steps that you can take to exploit WebSphere Real Time for RT Linux:

1. You can run a standard Java application in a real-time JVM to give you the benefit of Metronome garbage collection and achieve significant improvement in the predictability of the run time of your application.
2. Add the **-Xnojit** option after you have precompiled your code to use the ahead-of-time (AOT) compiler. See "Storing precompiled jar files into a shared class cache" on page 40.
3. Replace java.lang.Thread with javax.realtime.RealtimeThread in your application. You might see a slight improvement when compared with the AOT option.

   The main advantage of using real-time threads is the ability to control the priority that you give to each of the threads. Real-time threads can also be made periodic. To exploit these advantages, you must be prepared to make changes to the application itself.
4. Plan and write a specific application to use real-time threads and asynchronous event handlers to deal with timers or external events. Consider these three factors:
   - Planning the priority that you assign to your real-time threads
   - Deciding which memory areas you will use to hold objects
   - Communicating with the event handlers
5. Plan and write a specific application to use no-heap real-time threads. No-heap real-time threads are extensions of real-time threads and you have to consider the priority that you assign and the memory area. In general, take this step only if the application must handle events in times comparable to the GC pause time (sub-millisecond). Do not underestimate the complexity of developing with no-heap real-time threads.

Figure 7 on page 88 shows the steps described above.

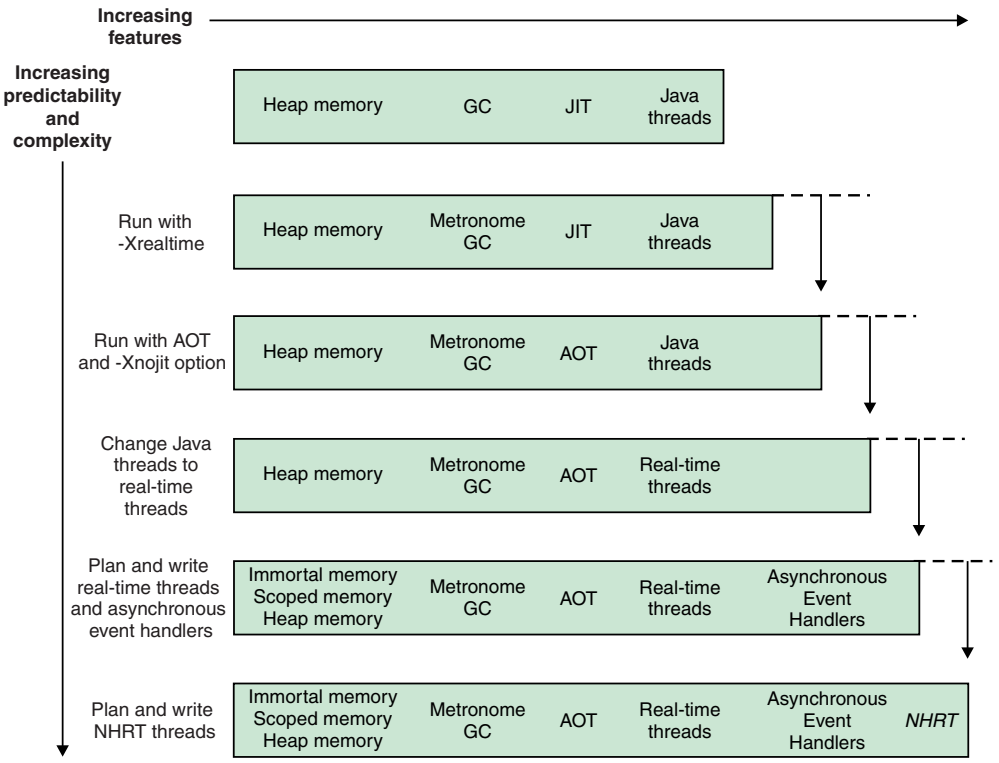


Figure 7. A comparison of the features of RTSJ with the increased predictability.

Planning your WebSphere Real Time for RT Linux application

When you are preparing to write real-time Java applications, you must consider whether to use Java threads, real-time threads, or no-heap real-time threads. In addition, you can decide which memory area that your threads will use.

About this task

When planning your application, these steps describe the decisions you need to make:

Procedure

1. Identify your tasks.
2. Decide the timing periods:
   - Responses greater than 10 ms, choose Java threads, just exploiting the Metronome Garbage Collector.

     These threads use only the heap memory for storage. Their disadvantage is that garbage collection interrupts your application but, because it is controlled by the Metronome Garbage Collector, the length and timing of the interruptions are predictable.
   - Responses less than 10 ms, choose real-time threads.

     Real-time threads can be placed in heap, scoped, or immortal memory. The benefits of using real-time threads are as follows:
     – They can run at a higher priority than standard Java threads.

– Garbage collection is under the control of the Metronome Garbage Collector. However, the garbage collector runs at a higher priority than the highest priority of a real-time thread and interrupts the running of your program.

- Responses less than a millisecond, choose no-heap real-time threads.

  The priority of no-heap real-time threads can be set higher than garbage collection and therefore is not interrupted significantly by the Metronome. Only the Metronome alarm thread runs at the top level of priority and that uses very small amounts of CPU.

3. Determine if your application requires asynchronous event handlers. This requirement depends on the structure of your program.
   - A time response less than 10 ms, choose real-time threads.
   - A time response less than a millisecond, choose no-heap real-time threads
4. Determine thread priorities. In general, the shorter the time period, the higher the priority.
5. Decide memory characteristics.
   - If a task has a variable or high allocation rate, which might overwhelm the GC, consider imposing a rate limit (using MemoryParameters) or consider allocating into a scoped memory area.
   - If a task generates a large amount of temporary data during a calculation, consider using a scoped memory area.
   - If a task generates some data during startup that is required for the lifetime of the JVM, consider using immortal memory. Try to avoid using immortal memory in cases where objects will continue to be created over the life of the JVM.
   - If tasks need to communicate, particularly if one is running under a no-heap real-time thread, consider using a scoped memory area for the communication.
   - If a task is running under a no-heap real-time thread, consider building a scoped memory area, for example LTMemory, to contain the no-heap thread, the runtime parameters, and possibly the wait-free queues that are used to communicate with the task. The LTMemory object must be built either in immortal or another scope to avoid errors when the no-heap thread attempts to reference it.
6. Modify the runtime options to improve the performance of your application, when you have decided the structure and content of your application. The next steps describe how to do this:
   a. During initial testing of your application, set generous amounts of space in heap, scoped, and immortal memory using the **-Xmx**, **-Xgc:immortalMemorySize=size** and **-Xgc:scopedMemoryMaximumSize=size** options.

      Note: With the Metronome GC the initial and maximum heap sizes must be the same because Metronome GC does not increase the size of the heap. Growing the heap is a nondeterministic operation.

   b. Use the **-verbose:gc** option to determine the amount of memory used.
   c. Modify the **-Xgc:targetUtilization** option to allow sufficient time for garbage collection to occur. The default is 70% and this percentage is usually adequate for most applications. Ensure that the garbage collection rate is slightly higher than the allocation rate.
   d. Set a realistic size for heap memory using the **-Xmx** option.

Related concepts

"Introduction to the Metronome Garbage Collector" on page 49
The benefit of the Metronome Garbage Collector is that the time it takes is more predictable and garbage collection can take place at set intervals over a period of time.

Modifying Java applications

To write code that makes use of the real-time Java features, use `javax.realtime.RealtimeThread` to replace `java.lang.Thread` for threads.

Before you begin

This example is based on JavaRadar.java class found in the `demo/realtime/sample_application.zip` file.

About this task

The programming model for real-time threads is similar to that for standard Java applications. However, this rather crude way of adding real-time threads to your programs does not take full advantage of the features of WebSphere Real Time for RT Linux. To do so, you must modify the threads so that they had a priority associated with them and also consider what memory areas they will use.

Just by changing the classes of your threads, you gain only a slight benefit for your application because the default priority of real-time threads is greater than that of standard Java threads.

To change JavaRadar to a RealtimeThread, you change the class it extends from `Thread` to `RealtimeThread`.

Replacement of `java.lang.Thread` by `javax.realtime.RealtimeThread`

The JavaRadar class in the sample application extends `java.lang.Thread`. For example:

```
public class JavaRadar extends Thread implements Radar
```

To make this Java thread a real-time thread, you redefine this class definition as follows:

```
public class RTJavaRadar extends RealtimeThread implements Radar
```

Writing real-time threads

So far, you have just modified an application; now it is time to write some code. You can write applications that use real-time threads to take advantage of the real-time priority levels and memory areas.

Before you begin

This example is based on JavaRadar.java, RTJavaRadar.java, and RTJavaControlLauncher.java classes found in the `demo/realtime/sample_application.zip` file.

This sample shows you how to use immortal memory with the same sample that is described in "Modifying Java applications" on page 90.

About this task

The programming model for real-time threads is similar to that for standard Java applications.

The benefits of using real-time threads are as follows:

- Full support for OS-level thread priorities on real-time threads.
- The use of scoped or immortal memory areas.
  - With scoped memory you can explicitly control the deallocation of memory without affecting Garbage Collection.
  - With no-heap real-time threads, you can use immortal memory to avoid garbage collection pauses.
  - Those real-time threads that reference objects in the heap are subject to garbage collection as are those real-time threads that are stored in the heap memory.
  - No-heap real-time threads cannot reference objects in heap memory and, as a consequence, they are not affected by garbage collection.

In Table 11, the priorities are assigned on the basis that the SimulationThread has the highest priority because it represents external events and must not be allowed to be preempted by anything in the program. The RadarThread needs to respond quickly to the pings from the controller. The quicker the response, the more accurate the measurement of the height of the lunar lander. The ListenThread also has to respond quickly to commands from the controller but takes second place to the RadarThread.

These three threads are in scoped memory because the simulation runs as a server. After it has run a simulation, it can exit the scoped memory area and then reenter it to wait for another run of the simulation. It is using scoped memory so that it can reset itself.

RTJavaRadarthread has the highest priority of the controller threads because it is more sensitive to timing because it is using this time to derive the height. It is immortal because it is running as a NHRT and the controller is run only once and the memory is released when the JVM exits.

For RTJavaControlThread and RTJavaEventThread, the time constraints are not as critical and therefore using heap memory is acceptable.

Finally, RTLoadThread performs no useful function for the lunar lander. However, it demonstrates that significant memory allocation and deallocation can be performed at a lower priority than other threads and not affect the performance of the higher priority threads of the lunar lander.

Table 11. Relationship of threads to memory areas in the sample application

Memory	Thread	Priority
Scoped	demo.sim.SimulationThread	38
	demo.sim.RadarThread	37
	demo.sim.SimulationThread.ListenThread	36
Immortal	demo.controller.RTJavaRadarThread	15

Table 11. Relationship of threads to memory areas in the sample application (continued)

Memory	Thread	Priority
Heap	demo.controller.RTJavaControlThread	14
	demo.controller.RTJavaEventThread	13
Scoped and Heap	demo.controller.RTLoadThread	12

Examples

This code from demo.sim.SimulationThread shows where the priority of 38 has been set. **1** This line of code retrieves the maximum priority that is available in the JVM.

```
super(null, area);

// Set priority separately, as we are using "this".
// Note that PriorityScheduler.MAX_PRIORITY has been deprecated.
this.setSchedulingParameters(new PriorityParameters(PriorityScheduler
        .getMaxPriority(this)));     1
```

This code from demo.sim.SimLauncher shows where scoped memory has been defined. **2** shows the allocation of LTMemory, which is a scoped memory area that allocates memory in linear time.

```
final IndirectRef<MemoryArea> myMemRef = new IndirectRef<MemoryArea>();

/*
 * The LTMemory object has to be created in a memory area that the
 * NHRTs can access.
 */
ImmortalMemory.instance().enter(new Runnable() {
    public void run() {
        myMemRef.ref = new LTMemory(10000000);  2
    }
});

final MemoryArea simMemArea = myMemRef.ref;
```

The ScopedMemoryArea object referenced by `simMemArea` is being allocated in immortal memory, because the NHRT must be able to reference the object that represents the ScopedMemoryArea. Allocating it on the heap results in the NHRT constructor throwing an IllegalArgumentException, because its memory area argument was on the heap.

```
simMemArea.enter(new Runnable() {
    public void run() {
        try {
            CommsControl commsControl = new CommsControl();
```

This code from demo.controller.RTJavaControlLauncher shows where immortal memory has been defined and used by RTJavaRadar. Because RTJavaRadar runs once during the whole lifetime of the controller JVM, it is designed to allocate memory only on startup; it can be safely run in immortal memory. The design of the application benefits because the Controller can access the RTJavaRadar methods without having to first enter the scoped memory area. Entering the scoped memory area is difficult because Controller was written to run in ordinary Java as well as in real-time Java.

```
final RadarPort radarPort = commsControl.getRadarPort();
EventPort eventPort = commsControl.getEventPort();

final IndirectRef<RTJavaRadar> radarRef = new IndirectRef<RTJavaRadar>();
```

```
// Create RTJavaRadar in Immortal, it is an NHRT.
// If it was in scoped, it's interaction with the other threads would
// be more complex.
ImmortalMemory.instance().enter(new Runnable() {
    public void run() {
        // Realtime version of Radar.
        radarRef.ref = new RTJavaRadar(radarPort, ImmortalMemory
                .instance());
    }
});

RTJavaRadar radarJava = radarRef.ref;
```

Related concepts

"Priorities and policies" on page 62
Regular Java threads, that is, threads allocated as `java.lang.Thread` objects, can use scheduling policies SCHED_OTHER, SCHED_RR or SCHED_FIFO. Real-time threads, that is, threads allocated as `java.lang.RealtimeThread`, and asynchronous event handlers use the SCHED_FIFO scheduling policy.

Writing asynchronous event handlers

Asynchronous event handlers react to timer events or to events that occur outside a thread; for example, input from an interface of an application. In real-time systems, these events must respond inside the deadlines that you set for your application.

Before you begin

This example is based on RTJavaEventThread.java and RTJavaControlLauncher.java classes found in the `demo/realtime/sample_application.zip` file.

About this task

In the sample application, the event thread waits on events from the simulation that signals a crash or a landing. In the real-time version of this thread, the AsyncEvent mechanism is used. These events are used to print out the appropriate status message and to cause the controller to exit.

The RTJava EventThread has two asynchronous events defined. They both have no parameters.

```
public class RTJavaEventThread extends RealtimeThread {

    private AsyncEvent landEvent = new AsyncEvent(),   Land
            crashEvent = new AsyncEvent();  Crash
```

These events create and register two asynchronous event handlers:

```
    /**
     * Pass a runnable object that will be fired when the land event occurs.
     *
     * @param runnable code to be executed when land event is triggered.
     */
    public void addLandHandler(Runnable runnable) {
        AsyncEventHandler handler = new AsyncEventHandler(runnable);
        this.landEvent.addHandler(handler);
    }

    /**
     * Pass a runnable object that will be run when the crash event occurs.
     *
```

```
     * @param runnable code to be executed when crash event is triggered.
     */
    public void addCrashHandler(Runnable runnable) {
        AsyncEventHandler handler = new AsyncEventHandler(runnable);
        this.crashEvent.addHandler(handler);
    }
```

When the crash or land messages are received, their corresponding asynchronous event handler is fired, causing its Runnable objects to be released.

```
tag = this.eventPort.receiveTag();

switch (tag) {
case EventPort.E_CRSH:
    // Crash
    this.crashEvent.fire();
    this.running = false;
    break;
case EventPort.E_LAND:
    // Land
    this.landEvent.fire();
    this.running = false;
    break;
}
```

Results

RTJavaControlLauncher.java contains invocations to the addLandHandler and addCrashHandler methods. The Runnable objects passed cause a message to be printed onto the console and the control thread is stopped when their associated asynchronous event handlers are fired. See RTJavaEventThread.java for the point where they are triggered.

```
// AEH runnable for land handler.
javaEventThread.addLandHandler(new Runnable() {
    public void run() {
        System.out.println("LAND!");
    }
});

// AEH runnable for crash handler.
javaEventThread.addCrashHandler(new Runnable() {
    public void run() {
        System.out.println("CRASH!");
    }
});
```

Writing NHRT threads

To add no-heap real-time threads (NHRT) to a Java application, use this tutorial to develop or modify your own programs.

Before you begin

This example is based on SimulationThread.java and SimLauncher.java classes found in the `demo/realtime/sample_application.zip` file.

About this task

The demo.sim.SimulationThread class is part of the simulation in the demo application. It is intended to act as a substitute for the real world, and, therefore, will probably run without interruption from the rest of the system. The thread is

created as a NoHeapRealtimeThread with the highest available priority, to ensure that the thread is not interrupted by garbage collection or by other threads on the system.

In SimulationThread, the following constructor calls the super constructor "NoHeapRealtimeThread(SchedulingParameters scheduling, MemoryArea area)", before then setting its SchedulingParameters and ReleaseParameters separately:

```
public SimulationThread(MemoryArea area, ControlPort controlPort,
            EventPort eventPort, RadarThread radarThread) {

        super(null, area);

        // Set priority separately, as we are using "this".
        // Note that PriorityScheduler.MAX_PRIORITY has been deprecated.
        this.setSchedulingParameters(new PriorityParameters(PriorityScheduler
                .getMaxPriority(this)));

        ReleaseParameters releaseParms = new PeriodicParameters(null,
                new RelativeTime(period, 0)); // 20ms cycle (50Hz)
        this.setReleaseParameters(releaseParms);

        // It is good practice to identify each of the threads.
        this.setName("SimulationThread");

        this.controlPort = controlPort;
        this.eventPort = eventPort;
        this.radarThread = radarThread;
    }
```

The other active threads in the simulation are also created as no-heap real-time threads (NHRTs), but of slightly lower priority. See "Writing real-time threads" on page 90 for the arrangement of the priorities.

The simulation has the option of running indefinitely, so that after a simulation has completed it restarts. Because the simulation is composed of NHRTs, you can choose ScopedMemory or ImmortalMemory. The sample application uses ScopedMemory for the simulation because it is appropriate to exit the ScopeMemoryArea that was allocated when the simulation finished and then reenter it to wait for the next run. In this case, no state is carried over from one run to the next.

Most classes are NHRT safe; however, most classes can be run in a manner that is not NHRT safe. For example, if the DatagramSockets were kept in immortal memory, or in an outer scoped memory area, problems might occur because they are not designed to span memory areas. The sample application uses just the one ScopedMemory area to prevent such problems.

Memory allocation in RTSJ

In RTSJ, you can allocate an object in a specific memory area in a number of ways and it is not always obvious which of these to choose at any given point.

Each approach has some characteristics, which vary between implementations of RTSJ, and make a difference to either the performance or the eventual memory footprint. This section outlines the available options and suggests occasions where they might be the most appropriate choice for allocating an object.

Static initializer

The simplest way to allocate an object in the immortal memory area is to allocate it in a static initializer. The advantage is that you do not have to deal with the issues of changing memory context, but the circumstances where this pattern is appropriate are quite limited. This approach is efficient in that the amount of immortal memory consumed is limited to that required for the object itself.

MemoryArea.newInstance(Class c)

This approach is straightforward if a thread is in a memory context and wants to allocate an object in another area, which must already be in the scope stack of the thread. The advantage is that you need access only to the class to be instantiated, but the newInstance method must build an appropriate constructor. This pattern is most appropriate if objects of a given class must be allocated infrequently, but otherwise tends to show high memory usage.

MemoryArea.newInstance(Constructor c, Object[] args)

Again, a simple approach if a thread is in a memory context and wants to allocate an object in another context, which must already be in the scope stack of the thread. In this case, you must pass a Constructor and some arguments and assumes the responsibility of ensuring that Constructor is valid in the current memory context. Because the newInstance method does not have to build a Constructor, the memory usage is lower than newInstance(Class c) and thus this pattern is more appropriate if objects are to be allocated more frequently and you are willing to pay the price of allocating the constructor in advance and storing it somewhere like ImmortalMemory.

MemoryArea.enter(Runnable r) followed by new <class>()

This approach makes the given MemoryArea the new default for allocations and removes the need for reflection and the attendant Constructor objects. Hence it is most appropriate if many objects are to be created because no additional memory usage occurs above the object itself. This approach works only if the desired area is not already active in the scope stack of any thread. The requirement to build a Runnable memory area makes this approach more complex than using newInstance because you generally should pass parameters either on the Runnable or through static or instance fields.

MemoryArea.executeInArea(Runnable r) followed by new <class>()

Again, this approach makes the given MemoryArea the new default for allocations and removes the need for reflection and the attendant Constructor objects. Hence it is most appropriate if many objects are to be created because no additional memory usage occurs above the object itself. You can use this approach if the desired area is already in the scope stack of the current thread and hence is more flexible than MemoryArea.enter. The requirement to build a Runnable makes this approach more complex than using newInstance because you generally should pass parameters either on the Runnable or through static or instance fields.

Class.newInstance()

This approach builds the new instance in the current memory area and therefore must be used with either MemoryArea.enter or executeInArea. No additional

memory usage occurs above the object itself.

Using the high-resolution timer

The real-time clock provides more precision that the clocks associated with the standard JVM.

Before you begin

This example is based on RTJavaRadar.java class found in the `demo/realtime/sample_application.zip` file.

About this task

Ordinary Java has limited ability for dealing with clocks and timers. The Real-Time Specification for Java allows absolute times to be specific with nanosecond precision and sufficient magnitude for wall-clock time. javax.realtime.HighResolutionTime and its subclasses are used to represent time with two components, milliseconds and nanoseconds.

WebSphere Real Time for RT Linux uses the support of the underlying operating system to supply the high resolution time. Current® Linux kernels supply a clock with, at best, 4 millisecond guaranteed precision. The Linux patches supplied with WebSphere Real Time for RT Linux provide a clock with a precision of closer to 1 microsecond.

The RTJavaRadar class demonstrates the use of the high-resolution timer:

- **1** gets the real-time clock.
- **2** gets the current absolute time.
- **3** gets the nanosecond component of time. The accuracy of the real-time clock means that using nanoseconds is reasonable.
- **4** gets the time before and after the ping.
- **5** returns the speed of descent of the lander.
- **6** makes the thread wait for 5 milliseconds before performing another iteration.

```
public void run() {
       // The following objects are created in advance and reused each
       // iteration.
       Clock rtClock = Clock.getRealtimeClock();         1
       AbsoluteTime time = rtClock.getTime();            2

       try {
           double height = 0.0, lastheight;
           long millis = time.getMilliseconds(), lastmillis;
           long nanos = time.getNanoseconds(), lastnanos;     3

           while (this.running) {

               lastmillis = millis;
               lastnanos = nanos;
               lastheight = height;

               // Rather than use the time = rtClock.getTime() form, this
               // method
               // replaces the values in a preexisting AbsoluteTime object.
               rtClock.getTime(time);                  4
               millis = time.getMilliseconds();
               nanos = time.getNanoseconds();
```

```
            // We time the time it takes to send the ping and receive the
            // pong.
            this.radarPort.ping();

            rtClock.getTime(time);                  4

            height = (time.getMilliseconds() - millis)
                    / demo.sim.RadarThread.timeScale;
            height += ((time.getNanoseconds() - nanos) / 1.0e6)       5
                    / demo.sim.RadarThread.timeScale;

            double difference = ((double) (millis - lastmillis)) / 1.0e3
                    + ((double) (nanos - lastnanos)) / 1.0e9;
            double speed = (height - lastheight) / difference;

            this.myHeight = height;
            this.mySpeed = speed;

            try {
                sleep(5);                       6
            } catch (InterruptedException e) {
                // This is not important.
            }
        }
```

The preceding code can be compared with the following standard JVM code in the JavaRadar class:

```
public void run() {
        try {
            double height = 0.0, lastheight;

            long nanos = System.nanoTime(), lastnanos;
            while (this.running) {
                /* Set the height every x milliseconds */
                Thread.sleep(5);
                lastnanos = nanos;
                lastheight = height;

                nanos = System.nanoTime();

                this.radarPort.ping();

                // Time scale is height units per millisecond
                height = ((System.nanoTime() - nanos) / 1.0e6)
                        / demo.sim.RadarThread.timeScale;

                double speed = (height - lastheight)
                        / (((double) (nanos - lastnanos)) / 1.0e9);

                this.myHeight = height;
                this.mySpeed = speed;
            }
```

Chapter 11. Using no-heap real-time threads

Metronome garbage collection provides more consistent response times, but sometimes it is appropriate to completely avoid interruptions from garbage collection.

NoHeapRealtimeThreads (NHRTs) are an extension to RealtimeThreads. They differ from real-time threads in that they do not have access to the heap memory. Without access to the heap, no-heap real-time threads (NHRTs) can continue to run even during a garbage collection cycle, with some restrictions. It follows that without access to the heap the programming model is different from the one for real-time threads.

Considerations when using NHRTs

Consider these points about NHRTs:

- The main reason for using NHRTs is with a task that cannot tolerate garbage collection. For example, if your application is time-critical and cannot tolerate any interruptions.
- If time is so critical that you are using NHRTs, also consider using the ahead-of time (AOT) compiler; that is, use the **-Xnojit** option.
- When you use the **-Xrealtime** option, you automatically use the Metronome Garbage Collector. The benefits of Metronome Garbage Collector might be sufficient for your enterprise, thus reducing the need to code NHRTs.
- NHRT threads run independently of the Garbage Collector because they have a priority higher than that of the Garbage Collector. Java threads can have a priority in the range 1 - 10. If NHRTs are present, the priority of Java threads is reset to 0 regardless of the priority set in your program. The Garbage Collector is automatically set to half a step higher than the highest real-time thread. You set the priority of your NHRTs to be at least one higher than the highest real-time thread. In this way, the NHRTs are independent of the garbage collector.

  Note: NHRTs are not entirely free of garbage collection because the Metronome alarm thread garbage collector runs at the highest priority in the system. This priority ensures that the JVM can be activated to check if the Garbage Collector needs to do anything. The work to run the Metronome alarm thread is very small and does not affect performance significantly. On a multi-processor system, the alarm thread can run simultaneously with NHRT threads and thus no garbage collection interruptions occur.
- Because NHRTs are restricted to the scoped and immortal memory areas, checks ensure that they are not allocated from the heap. The start method checks and returns an exception (`MemoryAccessError`) if NHRTs are allocated from the heap. NHRTs can access only `ImmortalMemory` and `ScopedMemory`.
- The semantics of locking are unchanged, so that NHRT threads can be blocked by normal threads if a lock is shared.
- A thread that is using the heap can have its priority boosted on a synchronized method when a NHRT tries to use the same method.
- Use nonblocking queues for communications between NHRTs and heap threads. Otherwise, separate the two types of threads.

Exceptions

These exceptions can occur when using NHRTs:

- `IllegalAssignmentError`. As an example, this error can occur when an attempt is made to create a reference to scoped memory in immortal memory.
- `MemoryAccessError`. As an example, this error can occur when a NHRT tries to reference heap memory.

Asynchronous event handling constraints

There multiple cases where Non Heap real-time can be blocked during garbage collection

1. When a Non Heap Real-Time calls fire() on an AsyncEvent which is already associated with handlers that are allocated from the Heap memory
2. When a Non Heap Real-Time calls setHandler() on an AsyncEvent which is already associated with handlers that are allocated from the Heap memory
3. When a Non Heap Real-Time calls addHandler() on an AsyncEvent which is already associated with handlers that are allocated from the Heap memory clean
4. When a Non Heap Real-Time calls destroy() on a Timer which is associated with handlers that are allocated from the Heap memory
5. When a Non Heap Real-Time calls start() on a Timer which is associated with handlers that are allocated from the Heap memory
6. When a Non Heap Real-Time calls stop() on a Timer which is associated with handlers that are allocated from the Heap memory
7. If a Non Heap Real-Time thread happens to be the last thread exiting a scope and is shutting down Timers/AsyncEvents from the scope that have associated handlers, that are allocated from the Heap memory

Make sure that Non Heap real time threads do no end up in this situation.

1. Avoid adding handlers allocated from the Heap to AsyncEvents, Timers that can potentially be fired by a Non Heap Real-Time thread.
2. Avoid conditions where a Non-Heap Realtime exits last from a scope which has AsyncEvents/Timers that have handlers associated from Heap.

Memory and scheduling constraints

The most obvious constraint on NoHeapRealtimeThreads (NHRT) is apparent in their name. The JVM prevents NHRTs loading references to objects on the heap onto its operand stack. To do so throws a javax.realtime.MemoryAccessError.

The JVM also guards against references to objects in scoped memory being stored in heap or immortal memory. Although scoped memory is not used exclusively by NHRTs, it is likely to be used if ImmortalMemory is inappropriate and memory deallocation is required in an NHRT context.

While an NHRT is executing, if it populates a field with a reference to an object, it can successfully overwrite any pre-existing reference to an object on the heap in that field. The pre-existing reference will be successfully overwritten by the NHRT without generating a MemoryAccessError.

Classloading constraints

Classes are loaded into the same memory areas as the classloader. The default for classloaders is ImmortalMemory.

In order for applications to provide the expected response times, they must be "warm". Applications should load their classes early so that class loading does not interrupt real-time threads and asynchronous event handlers later.

Constraints on Java threads when running with NHRTs

Because system properties are shared in a JVM and any thread can access the system properties, some care is required when using the getProperties and setProperties methods in JVMs in which NHRTs are run. For system properties to be accessible to NHRTs, they must be in immortal memory.

The `java.lang.System` class provides several methods that allow threads to interact with the system properties; including these methods:

```
String getProperty(String)
String getProperty(String,String)
Properties getProperties()

String setProperty(String,String)
void setProperties(Properties)
```

The real-time JVM uses an instance of the com.ibm.realtime.ImmortalProperties that was created specifically for the real-time JVM object to store all system properties. Use of this instance ensures that any calls to System.setProperty() or System.getProperties.setProperty() result in the property being stored in immortal memory. No special user code is required in this case but it is important to understand that each time a property is set some immortal memory is consumed.

Calls to setProperties() are a little more difficult because the shared Properties object is used to store system properties. When running in a real-time JVM that has NHRTs running, any call to setProperties must pass in an instance of an com.ibm.realtime.ImmortalProperties (or subclass) that was created in immortal memory. Use of this instance ensures that all properties that are set following setProperties are in immortal memory. Note that calling setProperties(null) results in a new ImmortalProperties object being created internally with a default set of properties, all of which consumes additional immortal memory.

Calls to getProperties() return either the object that was set or the default properties object, which is an com.ibm.realtime.ImmortalProperties object. To maximize compatibility with existing code that calls getProperties(), the ImmortalProperties object serializes the object and then deserializes in a standard JVM. The default behavior for the serialization of ImmortalProperties is to serialize a regular Properties object, because standard JVMs do not have the ImmortalProperties object and deserialization fails. To override this default behavior, ImmortalProperties provides the method enabledReplacement(boolean), which, if called with false, disables the default behavior. In this case, serialization serializes the ImmortalPropeties object and it is then possible to deserialize this and use the resulting object in a call to System.setProperties in a real-time JVM.

Note: Deserialization takes place in immortal memory, which might consume too much of this limited resource.

Security manager

The security manager set for the system is used by all types of threads in the JVM. For this reason, in a real-time JVM in which NHRTs run, the security manager must be allocated in immortal memory. The real-time JVM ensures that any security manager specified in the command-line options is allocated in immortal memory. The security manager can also be set through calls to `System.setSecurityManager(SecurityManager)`. If the application sets the security manager in this way, it must ensure that the security manager was allocated from Immortal so that NHRTs are able to run correctly.

The exceptions thrown and any objects returned by the security manager must either be in immortal memory, if cached, or be allocated in the current allocation context.

Synchronization

The MonitorControl class and its subclass PriorityInheritance manage synchronization, in particular priority inversion control. These classes allow the setting of a priority inversion control policy either as the default or for specific objects.

The WaitFreeReadQueue, WaitFreeWriteQueue, and WaitFreeDequeue classes allow wait-free communication between schedulable objects (especially instances of NoHeapRealtimeThread) and regular Java threads.

The wait-free queue classes provide safe, concurrent access to data shared between instances of NoHeapRealtimeThread and schedulable objects subject to garbage collection delays.

No-heap real-time class safety

In some circumstances, portions of the JSE API cannot necessarily be used in a no-heap context. Restrictions are placed on classes that are shared between heap and no-heap threads. Be aware of the classes supplied with the JVM that can be safely used.

Sharing objects

Methods that run in no-heap real-time threads throw a javax.realtime.MemoryAccessError whenever they try to load a reference to an object on a heap.

Figure 8 on page 103 is an example of the sort of code to be avoided:

```
/**
 * NHRTError1
 *
 * This example is a simple demonstration of an NHRT accessing
 * a heap object reference.
 *
 * The error generated is:
 *
 * Exception in thread "NoHeapRealtimeThread-0" javax.realtime.MemoryAccessError
 *      at NHRT.run(NHRTError1.java:56)
 *      at javax.realtime.RealtimeThread.runImpl(RealtimeThread.java:1754)

 */
import javax.realtime.*;

public class NHRTError1 {
    public static void main(String[] args) {
        NHRTError1 example = new NHRTError1();

        example.run();
    }

    public NHRTError1() {
        message = new String("This on the heap.");
    }

    static public String message; /* The NHRT can access static fields directly - they are always Immortal. */
    static public NHRT myNHRT = null;

    public void run() {
                ImmortalMemory.instance().executeInArea(new Runnable() {
            public void run() {
                NHRTError1.this.myNHRT = new NHRT();
            }
        });

        myNHRT.start();

        try {
            myNHRT.join();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
```

Figure 8. Example of an NHRT accessing a heap object reference

```
    /* A NHRT class */
    class NHRT extends NoHeapRealtimeThread {
        public NHRT() {
            super(null, ImmortalMemory.instance());
        }

        /* Prints the String via the static reference in NHRTError1.message */
        public void run() {
            System.out.println("Message: " + NHRTError1.message);
        }
    }

}
```

Figure 9. Example of an NHRT accessing a heap object reference (continued from Figure 1)

Figure 8 produces a javax.realtime.MemoryAccessError:

```
Exception in thread "NoHeapRealtimeThread-0" javax.realtime.MemoryAccessError
        at NHRTError1$NHRT.run(NHRTError1.java:56)
        at javax.realtime.RealtimeThread.runImpl(RealtimeThread.java:1754)
```

If an object is to be accessible to both a no-heap real-time thread and a standard Java thread, the object must be allocated in immortal memory. Similarly, if an

object is to be accessible to a no-heap real-time thread and a real-time thread, the object can also be held in a scoped memory area.

In Figure 8 on page 103, the reference to the String "This on the heap." was held in a class variable. This variable is accessible to NHRTs because all classes are allocated in immortal memory. Alternatively, the String could have been passed to the NHRTs constructor.

Most objects contain references to other objects, and so care must be taken when sharing such objects between ordinary threads and NHRTs. A typical example is of a LinkedList allocated in immortal memory, shared between an ordinary thread and a NHRT. If sufficient care is not taken, the standard thread might introduce objects into the LinkedList that are on the heap. Of greater concern is that the data structures that are allocated by the LinkedList to track objects are allocated on the heap by the ordinary thread, easily causing a MemoryAccessError in the NHRT.

Some classes cannot be safely shared between NHRTs and other threads, regardless of where individual instances of them might be allocated. These are classes that rely on objects stored in class variables, usually for caching purposes. InetAddress is a typical example that caches addresses; if the first thread to call certain methods in InetAddress is running on the heap, the same methods are unsafe to be called by NHRTs in the future.

Locking on objects with NHRTs

NHRTs must avoid synchronizing with other threads. Consider the following scenario:

- A real-time thread of low priority enters a synchronized block or method, synchronizing on an object.
- An NHRT of high priority is blocked when attempting to synchronize on the same object.
- Priority inheritance causes the real-time thread to temporarily assume the same priority as the NHRT.
- Garbage collection must then run at a higher priority than the NHRT and negates the reason for using the NHRT, which avoids interruption by garbage collection.

It is sometimes unavoidable that NHRTs and other threads synchronize on the same object, but you must minimize the possibility. Be careful to avoid unnecessary synchronization when sharing objects.

Restrictions on safe classes

Some considerations apply when an application contains both real-time thread and no-heap real-time thread objects.

- The no-heap real-time thread can suffer MemoryAccessErrors caused by interaction with the real-time thread.
- The no-heap real-time thread might be accidentally delayed by garbage collection caused by the real-time thread.

MemoryAccessErrors caused on a no-heap real-time thread

When the two types of thread both call methods on the same class, the real-time thread might “pollute” the static variables of the class with objects allocated from the heap. The no-heap real-time thread will receive a MemoryAccessError when

trying to access those heap objects. The pollution can also happen on instances of the class. Unfortunately, both problems are quite likely to be seen in typical coding patterns and so it is worth exploring a couple of cases.

If a class is performing a time-consuming operation, it often chooses to cache the result to improve performance of the subsequent operations. The cache is typically a collection such as a HashMap anchored in a static variable in the class. A real-time thread operating in heap context can store a heap object in this collection, which not only adds the object itself but also adds infrastructure objects to the collection; for example, parts of the index. When a no-heap real-time thread later tries to access the collection, even if it is not trying to access the object added by the other thread, it attempts to load the infrastructure objects and hence receive a MemoryAccessError. As class libraries develop and are tuned for performance, these caches become more common.

A class instance can also become polluted by heap objects in a variety of ways. Consider an instance built in immortal memory and thus accessible to both types of thread. If the first use of the object is by a realtime thread in heap context, you might find that a secondary object is stored in a field of the original object. If the secondary object is in heap context, subsequent use by the no-heap real-time thread again shows a MemoryAccessError. These secondary objects might not always be added on first use but after a number of uses, and might be designed to improve the performance of heavily used methods.

NoHeap thread delayed by garbage collection

No-heap threads must be assigned priorities that are higher than other threads to avoid being delayed by garbage collection.

Additionally, if a class contains any synchronized methods, it is possible that a no-heap real-time thread calling such methods might unintentionally be delayed by garbage collection. This scenario is described in “Locking on objects with NHRTs” on page 104.

If a class contains any synchronized methods (either static or instance methods), it is possible that a no-heap real-time thread calling such methods might unintentionally be delayed by garbage collection. The problem is caused if a real-time thread is accessing a synchronized method (static or instance) at the point where a no-heap real-time thread attempts to call another synchronized method that will block waiting for the other thread to complete. If the no-heap real-time thread has a higher priority than the other thread, you expect priority boosting of that thread to occur. If that thread is then forced to wait for a garbage collection interrupt, a priority inversion is possible, because the garbage collector thread has a priority higher than the highest priority real-time thread, which might not be as high as the no-heap real-time thread that is currently blocked waiting to enter the synchronized method.

The only way to fix such problems is to ensure that no-heap real-time threads never call synchronized methods on classes or instances that are shared with other thread types. Unfortunately, it is not always clear from a method signature if a method is synchronized; it might, for example, contain a synchronized block or call a synchronized method.

Summary

The NoHeapRealtimeThread class adds a significant amount of complexity to the real-time environment and a significant number of problems can be caused when a mixture of thread types operate in an environment. During development of an application, you must carefully design areas in which you have shared use of classes by the different thread types. Of particular importance is the use that these threads make of classes in the SDK. Because of the complexity of analysis, it is impossible to give any guarantee that all the classes provided in the SDK are safe for such shared use. Instead, a small subset of the classes have been verified. Initially, verification has concentrated on the MemoryAccessError aspect and the result is a list of classes that have been analyzed, tested, and modified where necessary to ensure that they can be used by both no-heap and other types of threads.

Safe classes

This section lists the set of classes that are intended to be safely used by NoHeapRealtimeThread and other thread types.

The main concern is focused on the MemoryAccessError aspect of safety. The following list details classes that can be used by all three thread types in the same JVM.

Note: Individual instances of the class might not always be safely shared.

For a class to be safely used by all thread types, the usage must initially obey some basic rules. The instance itself must be built in a memory area accessible to the thread intending to access it. If the class has public static fields, the user must avoid storing heap objects in them and the same limitation applies to any public instance fields.

Not all IBM-provided classes are NHRT-safe. The following list describes which classes are not safe. If a class is not listed, it is not safe.

java.lang package

Table 12. Classes in the java.lang package that are not NHRT-safe

Class	Method
java.lang.ProcessBuilder	*
java.lang.Thread	getAllStackTraces()Ljava.util.Map;
java.lang.ThreadGroup	*
java.lang.ThreadLocal	*
java.lang.InheritableThreadLocal	*

java.lang.reflect package

Table 13. Classes in the java.lang.reflect package that are not NHRT-safe

Class	Method
java.lang.reflect.Proxy.*	*

java.lang.ref package

All classes in the java.lang.ref package are NHRT-safe.

java.net package

Table 14. Classes in the java.net package that are not NHRT-safe

Class	Method
java.net.SocketPermission.*	newPermissionCollection()Ljava.net.SocketPermissionCollection;

java.io package

Table 15. Classes in the java.io package that are not NHRT-safe

Class	Method
java.io.ExpiringCache	*
java.io.SequenceInputStream	*
java.io.FilePermission	newPermissionCollection()Ljava.io.FilePermissionCollection;
java.io.ObjectInputStream	*
java.io.ObjectOutputStream	*
java.io.ObjectStreamClass	*

java.math package

Table 16. Classes in the java.math package that are not NHRT-safe

Class	Method
java.math.BigInteger	*

All other classes in the java.math package are NHRT-safe.

A class might be considered NHRT-safe, but might still not be suitable for use in an NHRT because it might have nondeterministic operation. Application developers must determine the real-time requirements of the classes that you use on a case-by-base basis, independent of whether or not a class is NHRT-safe.

The above list of packages is not intended to imply that subpackages are included and therefore safe. Hence, for example, the following classes are excluded from this safe list:

- java.lang.management.*
- java.lang.annotation.*
- java.lang.instrument.*

Chapter 12. Troubleshooting OutOfMemory Errors

Dealing with OutOfMemoryError exceptions, memory leaks, and hidden memory allocations

Related concepts

"Troubleshooting the Metronome Garbage Collector" on page 51
Using the command-line options, you can control the frequency of Metronome garbage collection, out of memory exceptions, and the Metronome behavior on explicit system calls.

Diagnosing OutOfMemoryErrors

Diagnosing OutOfMemoryError exceptions in Metronome Garbage Collector can be more complex than in a standard JVM because of the periodic nature of the garbage collector.

The characteristics of the different types of heap are described in "Memory management" on page 65. In general, an RTSJ application requires approximately 20% more heap space than a standard Java application.

By default, the JVM produces the following diagnostic output when an uncaught OutOfMemoryError occurs:

- A snap dump; see "Snap traces" on page 165.
- A Heapdump; see "Using Heapdump" on page 187.
- A Javadump; see "Using Javadump" on page 174

The dump file names are given in the console output:

```
JVMDUMP006I Processing dump event "systhrow", detail "java/lang/OutOfMemoryError" - please wait.
JVMDUMP007I JVM Requesting Snap dump using 'Snap.20081017.104217.13161.0001.trc'
JVMDUMP010I Snap dump written to Snap.20081017.104217.13161.0001.trc
JVMDUMP007I JVM Requesting Heap dump using 'heapdump.20081017.104217.13161.0002.phd'
JVMDUMP010I Heap dump written to heapdump.20081017.104217.13161.0002.phd
JVMDUMP007I JVM Requesting Java dump using 'javacore.20081017.104217.13161.0003.txt'
JVMDUMP010I Java dump written to javacore.20081017.104217.13161.0003.txt
JVMDUMP013I Processed dump event "systhrow", detail "java/lang/OutOfMemoryError".
```

The Java backtrace shown on the console output, and also available in the Javadump, indicates where in the Java application the OutOfMemoryError occurred. The next step is to find out which RTSJ memory area is full. The JVM memory management component issues a tracepoint that gives the size, class block address, and memory space name of the failing allocation. You find this tracepoint in the snap dump:

```
<< lines omitted... >>
09:42:17.563258000 *0xf2888e00       j9mm.101  Event       J9AllocateIndexableObject() returning NULL! 80 bytes requested
object of class 0xf1632d80 from memory space 'Metronome' id=0xf288b584
```

The tracepoint ID and data fields might vary from that shown, depending on the type of object being allocated. In this example, the tracepoint shows that the

allocation failure occurred when the application attempted to allocate a 33.6 MB object of type `class 0x81312d8` in the `Metronome heap`, memory segment `id=0x809c5f0`.

You can determine which RTSJ memory area is affected by looking at the memory management information in the Javadump:

```
NULL           ------------------------------------------------------------------------
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
NULL
1STMEMTYPE     Object Memory
NULL           region     start      end        size       name
1STHEAP        0xF288B584 0xF2A1C000 0xF6A1C000 0x04000000 Default
NULL
1STMEMUSAGE    Total memory available: 67108864 (0x04000000)
1STMEMUSAGE    Total memory in use:    66676824 (0x03F96858)
1STMEMUSAGE    Total memory free:      00432040 (0x000697A8)
NULL
NULL           region     start      end        size       name
1STHEAP        0xF288B5A4 0xF17FF008 0xF27FF008 0x01000000 Immortal
NULL
1STMEMUSAGE    Total memory available: 16777216 (0x01000000)
1STMEMUSAGE    Total memory in use:    00450816 (0x0006E100)
1STMEMUSAGE    Total memory free:      16326400 (0x00F91F00)
NULL
1STSEGTYPE     Internal Memory
NULL           segment    start      alloc      end        type       size
1STSEGMENT     0x0808DA48 0x0814A0A8 0x0814A0A8 0x0815A0A8 0x01000040 0x00010000
1STSEGMENT     0x0808DB50 0x08131EB8 0x08131EB8 0x08141EB8 0x01000040 0x00010000
<< lines removed for clarity >>
```

You can determine the type of object being allocated by looking at the classes section of the Javadump:

```
NULL           ------------------------------------------------------------------------
0SECTION       CLASSES subcomponent dump routine
NULL           ================================
<< lines omitted... >>
1CLTEXTCLLOD    ClassLoader loaded classes
2CLTEXTCLLOAD    Loader *System*(0xF182BB80)
<< lines omitted... >>
3CLTEXTCLASS      [C(0xF1632D80)
```

Information in the Javadump confirms that the attempted allocation was for a character array, in the normal heap (`ID=0xF288B584`) and that the total allocated size of the heap, indicated by the appropriate `1STHEAP` line, is 67108864 decimal bytes or 0x04000000 hex bytes, or 64 MB.

In this example, the failing allocation is large in relation to the total heap size. If your application is expected to create 33 MB objects, the next step is to increase the size of the heap, using the **-Xmx** option.

It is more common for the failing allocation to be small in relation to total heap size. This is because of previous allocations filling up the heap. In these cases, the next step is to use the Heapdump to investigate the amount of memory allocated to existing objects.

The Heapdump is a compressed binary file containing a list of all objects with their object class, size, and references. Analyze the Heapdump using the Memory Dump Diagnostics for Java tool (MDD4J), which is available for download from the IBM Support Assistant (ISA).

Using MDD4J, you can load a Heapdump and locate tree structures of objects that are suspected of consuming large amounts of heap space. The tool provides various views for objects on the heap, Figure 10 on page 112 shows a view created by MDD4J detailing likely leak suspects, and giving the top five objects and packages contributing to the heap size.

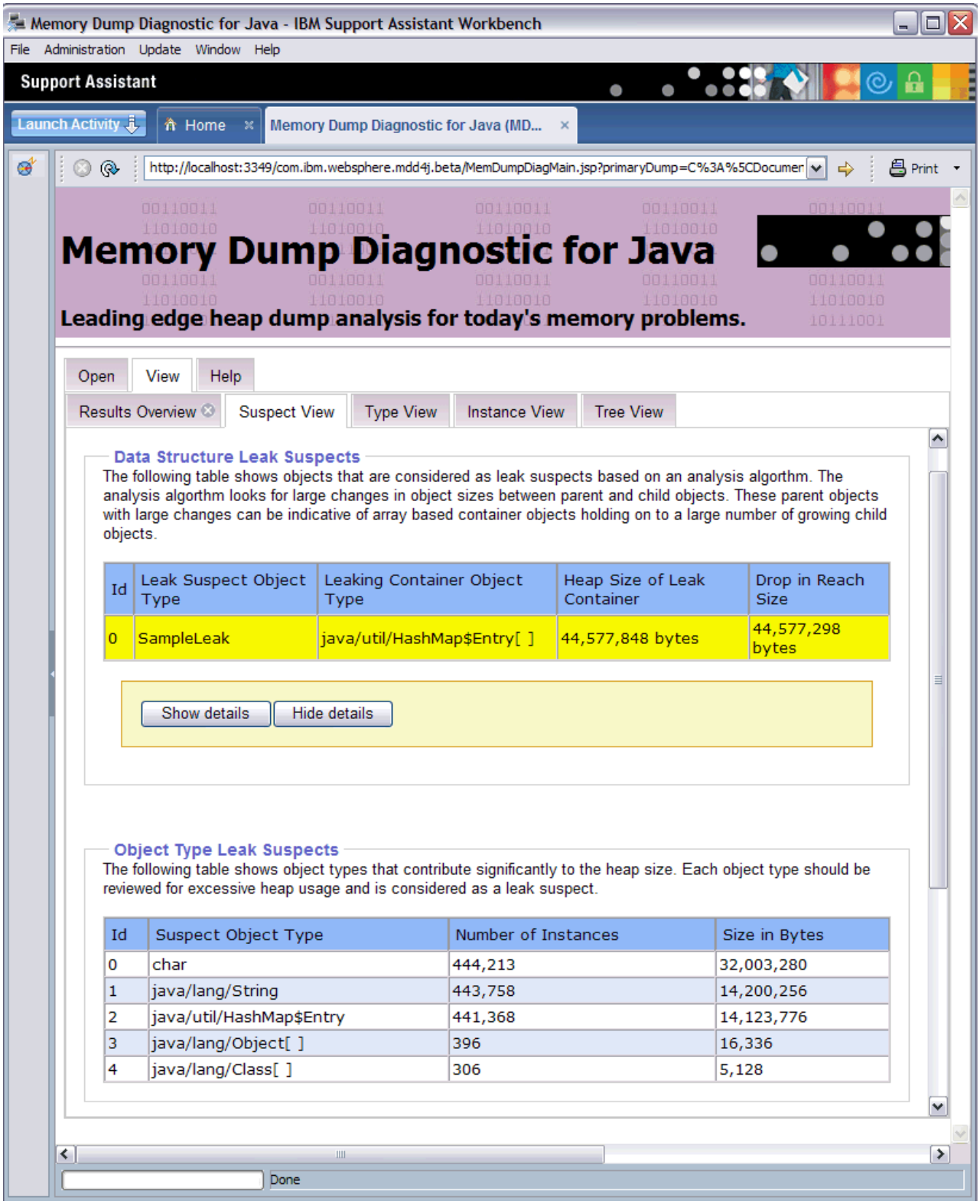


Figure 10. MDD4J has analyzed the heapdump and determined that there is a leak suspect

Selecting the tree view gives us further information about the nature of the leaking container object.

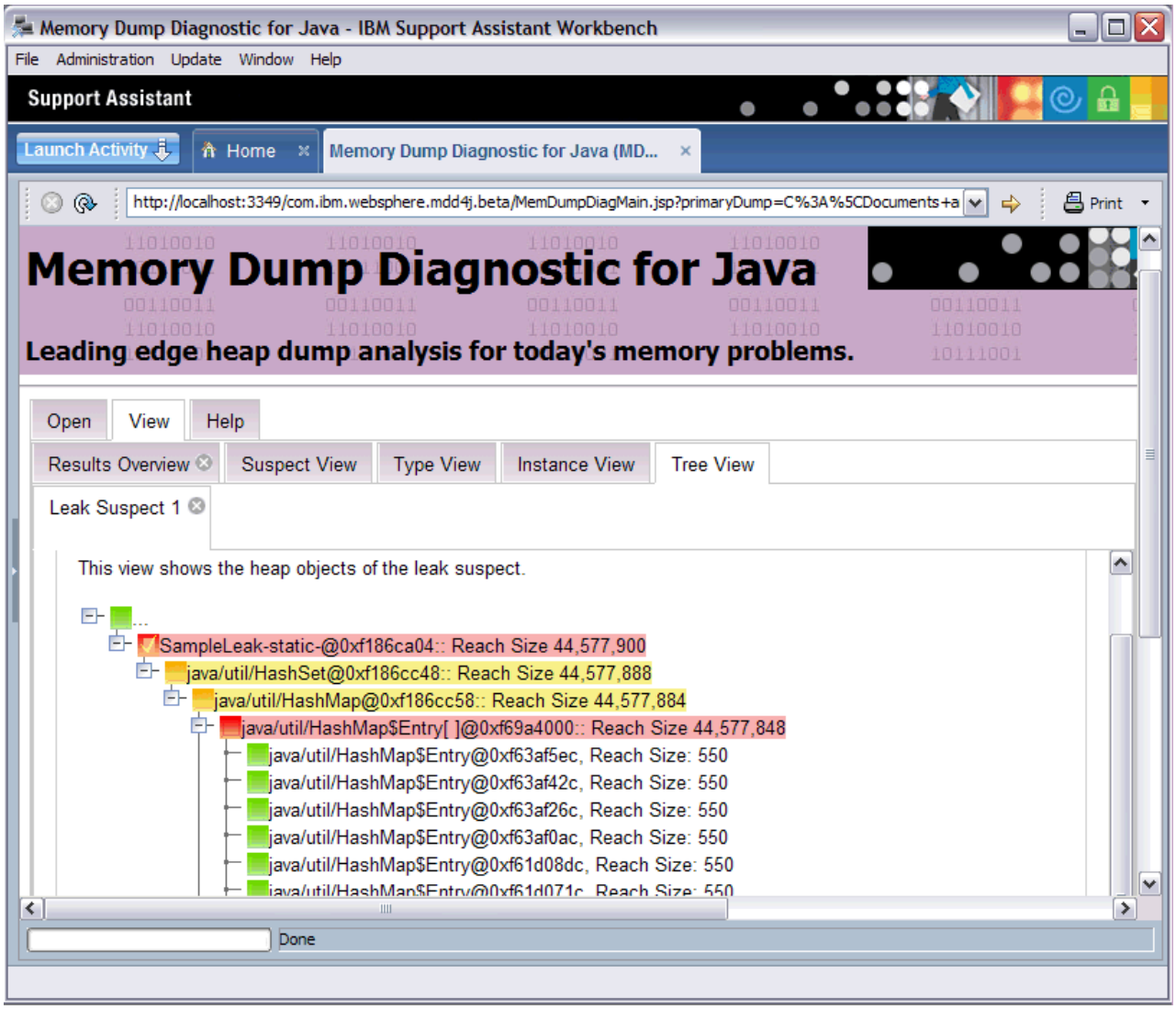


Figure 11. MDD4J shows the heap objects of the leak suspect

By default, a single Heapdump file containing all objects in all RTSJ memory spaces is produced. Use the command-line option **-Xdump:heap:request=multiple** to request a separate Heapdump for each memory space. With multiple dumps, you can examine just the set of objects that are allocated in a specific memory area. You identify the Heapdumps by the file name given on the console output:

```
JVMDUMP006I Processing Dump Event "uncaught", detail "java/lang/OutOfMemoryError" - Please Wait.
<< lines omitted... >>
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Default0809DCD8-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Default0809DCD8-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Immortal0809DCF4-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Immortal0809DCF4-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Scope0809DD10-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Scope0809DD10-0002.phd
<< lines omitted... >>
JVMDUMP013I Processed Dump Event "uncaught", detail "java/lang/OutOfMemoryError".
Exception in thread "RTJ Memory Consumer (thread_type=Realtime)" java.lang.OutOfMemoryError
 at tests.com.ibm.jtc.ras.runnable.DepleteMemory.depleteMemory(DepleteMemory.java:57)
<< lines omitted... >>
```

How the IBM JVM manages memory

The IBM JVM requires memory for several different components, including memory regions for classes, compiled code, Java objects, Java stacks, and JNI stacks. Some of these memory regions must be in contiguous memory. Other memory regions can be segmented into smaller memory regions and linked together.

Dynamically loaded classes and compiled code are stored in segmented memory regions for dynamically loaded classes. Classes are further subdivided into writable memory regions (RAM classes) and read-only memory regions (ROM classes). At runtime, the class cache is memory mapped, but not necessarily loaded, into a contiguous memory region on application startup. As classes are referenced by the application, classes and compiled code in the class cache are mapped into storage. The ROM component of the class is shared between multiple processes referencing this class. The RAM component of the class is created in the segmented memory regions for dynamically loaded classes when the class is first referenced by the JVM. AOT-compiled code for the methods of a class in the class cache are copied into an executable dynamic code memory region, because this code is not shared by processes. Classes that are not loaded from the class cache are similar to cached classes, except that the ROM class information is created in segmented memory regions for dynamically loaded classes. Dynamically generated code is stored in the same dynamic code memory regions that hold AOT code for cached classes.

All Java objects are stored in the standard heap memory when running the JVM without the **-Xrealtime** option. If the **-Xrealtime** option is used, objects can also be allocated out of two additional memory regions called immortal memory and scoped memory.

The stack for each Java thread can span a segmented memory region. The JNI stack for each thread occupies a contiguous memory region.

To determine how your JVM is configured, run with the **-verbose:sizes** option. This option prints out information about memory regions where you can manage the size. For memory regions that are not contiguous, an increment is printed describing how much memory is acquired every time the region needs to grow.

Here is example output using the **-Xrealtime -verbose:sizes** options:

```
-Xmca32K                        RAM class segment increment
-Xmco128K                       ROM class segment increment
-Xms64M                         initial memory size-Xgc:immortalMemorySize=16M     immortal memory sp
-Xgc:scopedMemoryMaximumSize=8M scoped memory space maximum size
-Xmx64M                         memory maximum
-Xmso256K                       operating system thread stack size
-Xiss2K                         java thread stack initial size
-Xssi16K                        java thread stack increment
-Xss256K                        java thread stack maximum size
```

This example indicates that the RAM class segment is initially 0, but grows by 32 KB blocks as required. The ROM class segment is initially 0, and grows by 128 KB blocks as required. You can use the **-Xmca** and **-Xmco** options to control these sizes. RAM class and ROM class segments grow as required, so you will not typically need to change these options.

The immortal memory is a contiguous region and might need to be preallocated a larger space. In this example, the immortal memory region is preallocated at 16

MB. If you try to write more than 16 MB of objects into this immortal memory region, you receive an OutOfMemory exception because, by definition, this memory area is not garbage collected.

The scoped memory region is contiguous and in this example is pre-allocated at 8 MB. If you have many scoped memory areas active when the program is running, you might need to specify a larger scoped memory region.

Use the admincache utility to determine how large your memory mapped region will be if you use the class cache. Here is a sample of the output from the command `admincache -Xrealtime -printStats -nologo`:

```
J9 Java(TM) admincache 1.0

Current statistics for cache "sharedcc_localuser":


base address       = 0xA52B4000
end address        = 0xA59B7000
allocation pointer = 0xA59B4000

cache size         = 7356040
free bytes         = 330604
ROMClass bytes     = 3798460
AOT bytes          = 3101560
Data bytes         = 3812
Metadata bytes     = 121604
Metadata % used    = 1%

# ROMClasses       = 1044
# AOT Methods      = 1652
# Classpaths       = 2
# URLs             = 1
# Tokens           = 0
# Stale classes    = 0
% Stale classes    = 0%

Cache is 95% full
```

The cache size indicates that the memory mapped region will be slightly over 7 MB in space. The ROM class and AOT bytes take up the majority of this space, using slightly over 3 MB each.

Example OutOfMemoryError in immortal memory space

This example shows how to identify an OutOfMemoryError in the immortal memory space and describes steps to take to prevent the problem.

The snap dump shows that two small allocation requests have failed in the immortal memory area id=0x809dd1c:

```
16:08:04.876087000 083d4000       j9mm.100  Event       J9AllocateObject() returning NULL!
  16 bytes requested for object of class 0x8110e60 from memory space 'Immortal' id=0x809dd1c
16:08:04.876171000 083d4000       j9mm.100  Event       J9AllocateObject() returning NULL!
  32 bytes requested for object of class 0x81180f0 from memory space 'Immortal' id=0x809dd1c
```

The Javadump shows that the immortal memory space is full:

```
NULL           ------------------------------------------------------------------------
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
1STHEAPFREE    Bytes of Heap Space Free: 3f0c000
1STHEAPALLOC   Bytes of Heap Space Allocated: 4000000
1STHEAPFREE    Bytes of Immortal Space Free: 0
```

```
1STHEAPALLOC   Bytes of Immortal Space Allocated: 1000000
<< lines omitted... >>
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Immortal Segment  ID=0809DD1C
1STSEGMENT     0809D510 B279D008 B379D008 B379D008  00001008 1000000
```

Figure 12 is a screen capture of an MDD4J analysis showing that a very large LinkedList has been allocated, consuming a large proportion of the available memory.

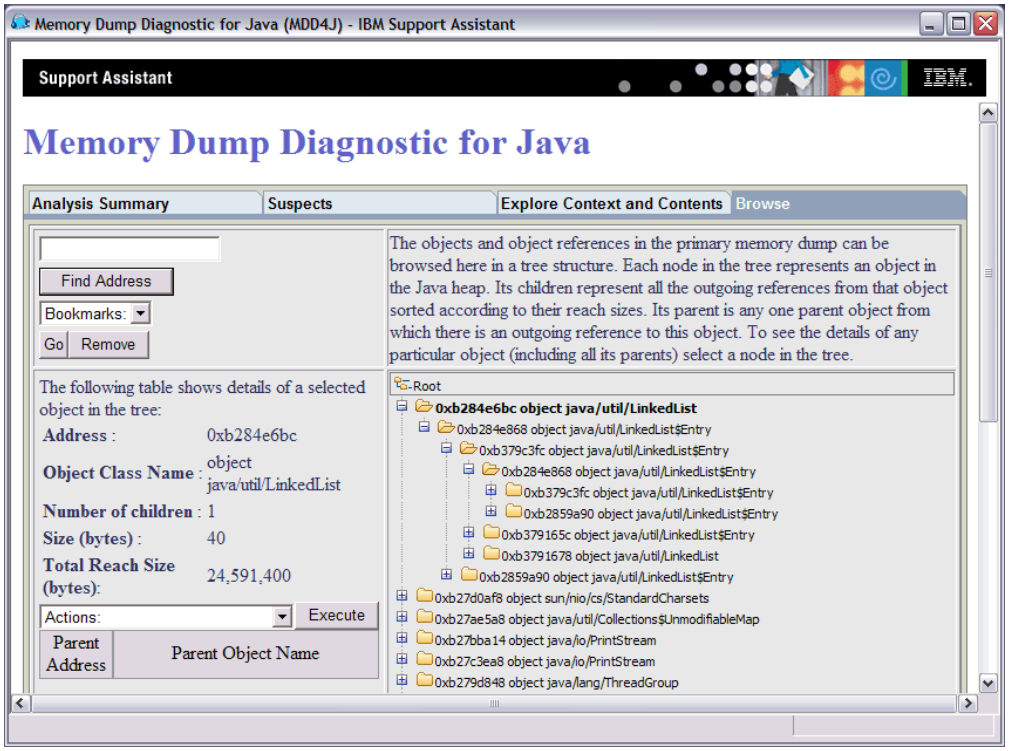


Figure 12. MDD4J showing a large linked list

You are recommended to minimize the number of objects allocated in the immortal memory area because objects in the immortal area are not garbage collected. The most common immortal memory usage is class loading, which is a finite activity occurring mostly during JVM and application initialization. Applications with high numbers of loaded classes (or other immortal memory usage) can increase the size of the immortal memory area using the **-Xgc:immortalMemorySize=<size>** option. The default size for the immortal memory area is 16 MB.

If increasing the size of the immortal memory area only delays the OutOfMemoryError for immortal memory, investigate the pattern of continued allocation of immortal data, either related to class loading or other application objects.

Example OutOfMemoryError in scoped memory space

This example shows you how to identify an OutOfMemoryError in scoped memory space and describes steps to take to prevent the problem.

Use the command-line option **-Xdump:heap:request=multiple** to produce separate dumps for each memory space:

```
VMDUMP006I Processing Dump Event "uncaught", detail "java/lang/OutOfMemoryError" - Please Wait.
JVMDUMP007I JVM Requesting Snap Dump using '/home/test/snap-0001.trc'
JVMDUMP010I Snap Dump written to /home/test/snap-0001.trc
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Default0809DCD8-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Default0809DCD8-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Immortal0809DCF4-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Immortal0809DCF4-0002.phd
JVMDUMP007I JVM Requesting Heap Dump using '/home/test/heapdump-Scope0809DD10-0002.phd'
JVMDUMP010I Heap Dump written to /home/test/heapdump-Scope0809DD10-0002.phd
JVMDUMP007I JVM Requesting Java Dump using '/home/test/javacore-0003.txt'
JVMDUMP010I Java Dump written to /home/test/javacore-0003.txt
JVMDUMP013I Processed Dump Event "uncaught", detail "java/lang/OutOfMemoryError".
Exception in thread "RTJ Memory Consumer (thread_type=Realtime)" java.lang.OutOfMemoryError
   at tests.com.ibm.jtc.ras.runnable.DepleteMemory.depleteMemory(DepleteMemory.java:57)
   at tests.com.ibm.jtc.ras.runnable.DepleteMemory.run(DepleteMemory.java:26)
<< lines omitted... >>
```

The snap dump shows that two allocation requests have failed in scoped memory area id=0x809dd10:

```
16:14:45.887176823 08480900       j9mm.100  Event       J9AllocateObject() returning NULL!
  16 bytes requested for object of class 0x8110e38 from memory space 'Scoped' id=0x809dd10
16:14:45.887252747 08480900       j9mm.100  Event       J9AllocateObject() returning NULL!
  32 bytes requested for object of class 0x81180c8 from memory space 'Scoped' id=0x809dd10
```

The Javadump shows that for the scoped memory area with id=0x809dd10, the allocated size of the memory area is quite small, only 60 KB; in this case, increase the size of the scoped memory area in the application code.

```
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
1STHEAPFREE    Bytes of Heap Space Free: 3eb0000
1STHEAPALLOC   Bytes of Heap Space Allocated: 4000000
1STHEAPFREE    Bytes of Immortal Space Free: f47474
1STHEAPALLOC   Bytes of Immortal Space Allocated: 1000000
1STHEAPFREE    Bytes of Scoped Space ID=0809DD10 Free: eb00
1STHEAPALLOC   Bytes of Scoped Space Allocated: eb00
.......
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Scoped Segment  ID=0809DD10
1STSEGMENT     0809D560 08416350 08424E50 08424E50  00002008 eb00
1STSEGSUBTYPE  Immortal Segment  ID=0809DCF4
1STSEGMENT     0809D4E8 B2857008 B3857008 B3857008  00001008 1000000
```

In the example Javadump, the scoped memory area appears to be empty. It appears empty because the Javadump is produced when the OutOfMemoryError reaches the JVM, at which time the scope has been exited and cleaned up. You can produce a Javadump at the point of failure by using the **-Xdump:java:events=throw,filter=java/lang/OutOfMemoryError** command-line option. By using this option, the free space in the scoped memory area is correctly reported.

It is also possible to exhaust the total space available for scoped memory; in this case, increase the size of the scoped memory area using the command-line option **-Xgc:scopedMemoryMaximumSize=<size>**. The default size for the scoped memory area is 8 MB. If the total space available for scoped memory becomes exhausted, you see different messages on the console; for example:

```
Exception in thread "main" java.lang.OutOfMemoryError: Creating (LTMemory) Scoped memory # 0 size=16777216
        at javax.realtime.MemoryArea.create(MemoryArea.java:808)
        at javax.realtime.MemoryArea.create(MemoryArea.java:798)
        at javax.realtime.ScopedMemory.create(ScopedMemory.java:1359)
        at javax.realtime.ScopedMemory.create(ScopedMemory.java:1351)
        at javax.realtime.ScopedMemory.initialize(ScopedMemory.java:1705)
        at javax.realtime.ScopedMemory.<init>(ScopedMemory.java:216)
        at javax.realtime.ScopedMemory.<init>(ScopedMemory.java:164)
```

Diagnosing problems in multiple heaps

You can use the address ranges provided in the Javadump with the occupancy information in the Heapdump to help analyze OutOfMemoryErrors in multiple RTSJ memory areas.

In this Javadump, the immortal segment ranges from 0xB281C008 to 0xB381C008, and the normal heap segment ranges from 0xB381D008 to 0xB781D008:

```
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
1STHEAPFREE    Bytes of Heap Space Free: 58000
1STHEAPALLOC   Bytes of Heap Space Allocated: 4000000
1STHEAPFREE    Bytes of Immortal Space Free: b319d8
1STHEAPALLOC   Bytes of Immortal Space Allocated: 1000000
NULL
1STSEGTYPE     Internal Memory
<< lines omitted... >>
1STSEGTYPE     Object Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGSUBTYPE  Immortal Segment  ID=0809C68C
1STSEGMENT     0809BE80 B281C008 B381C008 B381C008  00001008 1000000
1STSEGSUBTYPE  Heap Segment  ID=0809C670
1STSEGMENT     0809BE08 B381D008 B781D008 B781D008  00000009 4000000
NULL
1STSEGTYPE     Class Memory
NULL           segment  start    alloc    end       type     bytes
1STSEGMENT     08158154 083FFD68 083FFEF0 08407D68  00010040 8004
```

The Heapdump is a compressed binary file containing a list of all objects with their object class, size, and references. Analyze the Heapdump using the Memory Dump Diagnostics for Java tool (MDD4J) which is available for download from the IBM Support Assistant (ISA).

You can use the object memory locations listed by MDD4J to determine the memory space in which an object is. In Figure 13 on page 119, the first 13 roots objects have addresses in the 0xB28*nnnnn* range, which shows that they are in the immortal memory area. The last root object and the Integer objects that it contains are in the 0xB61*nnnnn* range, which shows that they are in the normal heap.

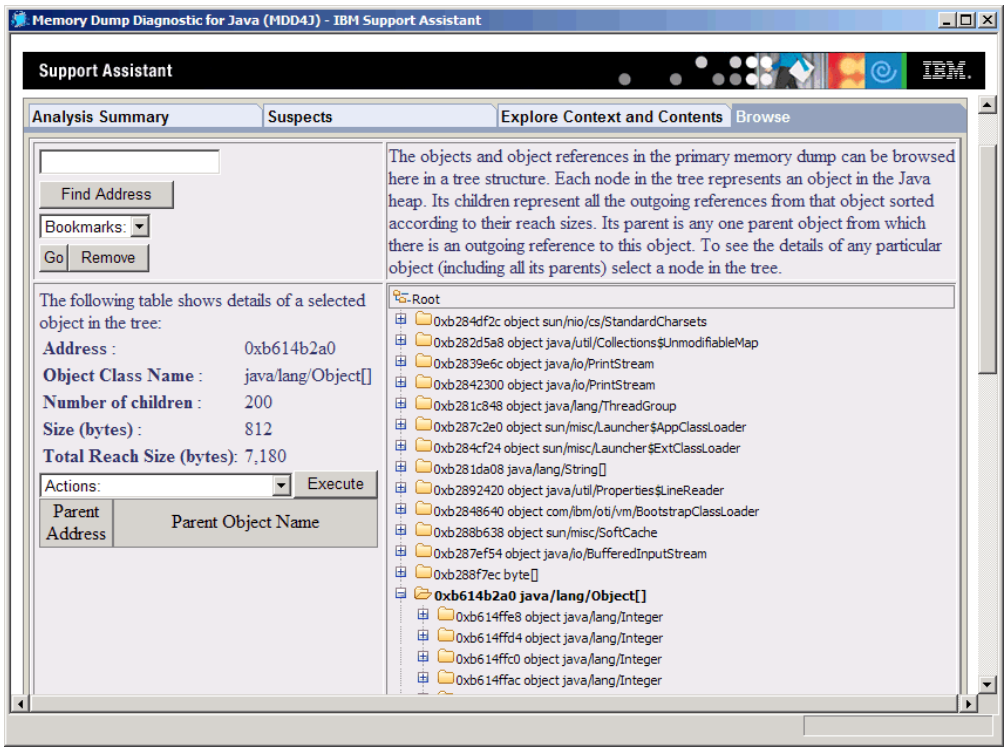


Figure 13. MDD4J showing objects in both the immortal and normal heap memory areas

Avoiding memory leaks

Immortal and scoped memory are not subject to garbage collection. In the case of immortal memory, the memory is freed only when the JVM exits. Scoped memory areas are freed only after their reference counts go to zero. Long-running tasks running in these contexts must be written in such a way that, after the task has warmed up, no additional memory from the immortal memory area is allocated.

Loading classes uses a small amount of immortal memory. These classes are not garbage collected in the real-time environment. As such, loading classes that are not required by the application can cause the application to use more immortal memory than necessary.

If your application contains classes that implement the Serializable interface, adjust the initial immortal memory size to account for the footprint of generated classes. Each constructor has one generated object per class, in the form of "GeneratedSerializationConstructorAccessorXXX" (whereXXX is a number) that is loaded into immortal memory the first time the object is serialized.

Objects allocated out of the immortal memory area cannot be garbage collected and, as such, you must be very careful about which objects are allocated from the immortal memory area and try to minimize usage as much as possible. Consider object pooling in immortal memory if the immortal memory area is being used more than occasionally.

Hidden memory allocation through language features

In a scoped or immortal memory context, avoid the variable arguments language feature because they allocate hidden memory.

Variable arguments (vararg)

The Java language implements variable arguments by passing them to the method as an array. The compiler makes calling variable argument methods easy by creating and initializing the array for you.

Memory can be lost by calling a variable argument method in an immortal or scoped memory context. Do not use variable arguments in scoped or immortal memory contexts. Instead, explicitly create an array and use it in place of the variable arguments.

Here is an example showing two equivalent ways of calling a variable argument method:

```
public class VarargEx {

    public static void main(String[] args) {
        System.out.println("Sum: "+ sum(1.0, 2.0 , 3.0, 4.0));

        /* ... is the same as ... */

        double array[] = new double[4];

        array[0] = 1.0; array[1] = 2.0; array[2] = 3.0; array[3] = 4.0;
        System.out.println("Sum: " + sum(array));
    }

    static double sum(double... params) {
        double total=0.0;

        for(double num : params) {
            total += num;
        }

        return total;
    }
}
```

String concatenation

Adding to a string to produce a longer string is implemented using java.lang.StringBuilder objects, which requires memory allocations.

Autoboxing

Autoboxing involves creating an object to contain a basic type, which requires memory allocations.

Using reflection across memory contexts

If a constructor object has been built in a scoped memory area, it can be used only in the same scope or an inner scope. Any attempt to use that constructor object in immortal, heap, or an outer scope memory context will fail.

The exception thrown when reflection has occurred across memory contexts will be similar to the following:

```
Exception in thread "NoHeapRealtimeThread-14" javax.realtime.IllegalAssignmentError
   at java.lang.reflect.Constructor$1.<init>(Constructor.java:570)
   at java.lang.reflect.Constructor.acquireConstructorAccessor(Constructor.java:568)
   at java.lang.reflect.Constructor.newInstance(Constructor.java:521)
   at testMain$TestRunnable$1.run(testMain.java:40)
   at javax.realtime.MemoryArea.activateNewArea(MemoryArea.java:597)
   at javax.realtime.MemoryArea.doExecuteInArea(MemoryArea.java:612)
   at javax.realtime.ImmortalMemory.executeInArea(ImmortalMemory.java:77)
   at testMain$TestRunnable.allocate(testMain.java:36)
   at testMain$TestRunnable.run(testMain.java:12)
   at java.lang.Thread.run(Thread.java:875)
   at javax.realtime.ScopedMemory.runEnterLogic(ScopedMemory.java:280)
   at javax.realtime.MemoryArea.enter(MemoryArea.java:159)
   at javax.realtime.ScopedMemory.enterAreaWithCleanup(ScopedMemory.java:194)
   at javax.realtime.ScopedMemory.enter(ScopedMemory.java:186)
   at javax.realtime.RealtimeThread.runImpl(RealtimeThread.java:1824)
```

It is possible to work around this restriction by using the constructor in the same scope as it was allocated.

Using inner classes with scoped memory areas

When using inner classes in the context of scoped memory areas, you must take care when instantating inner class objects if the outer and inner objects are located in different memory areas. An IllegalAssignmentError will result from compiler-generated code that is not visible in the original source code, if the inner object is not capable of storing a reference to the outer object.

An inner class object must be able to store an implicit reference to its outer class object. If the reference violates RTSJ memory reference rules then an IllegalAssignmentError will be generated.

Most inner classes (including local and anonymous inner classes) will contain a compiler-generated (synthetic) non-static field for the instance of the lexically enclosing outer class. The only exception occurs when an inner class instance does not have an enclosing outer object, such as an anonymous class object instantiated in a static intializer block. The synthetic field of the inner object will contain a reference to the outer object. This is implemented by the compiler as a convenience to the java programmer. The field will not be visible in the original source code, although it is possible to write similar code using static nested classes with a reference that is visible. If the implicit reference violates RTSJ memory area rules, then an IllegalAssignmentError will be thrown, when the inner object is contructed, as it attempts to store the reference to the outer object.

In general, you cannot violate RTSJ memory reference rules when using inner classes. You cannot create an inner object if a reference to the associated outer object violates RTSJ memory reference rules. This rule means that an inner object allocated in immortal or heap cannot have a reference to an outer object from scoped memory. An inner object from scoped memory can have a reference to an outer object from scoped memory, but the outer object must be allocated from the same scoped memory area or an outer scoped memory area.

There are work-arounds, including:

- use static nested classes to eliminate the implicit reference
- choose memory areas to ensure inner and outer object relationships do not violate memory area reference restrictions

Chapter 13. Problem determination

This section describes problem determination. It is intended to help you find the kind of fault you have and from there to do one or more of the following tasks:

- Fix the problem
- Find a good workaround
- Collect the necessary data with which to generate a bug report to IBM

The chapters in this part are:

First steps in problem determination

Before proceeding in problem determination, there are some initial questions to be answered.

Have you changed anything recently?
If you have changed, added, or removed software or hardware just before the problem occurred, back out the change and see if the problem persists.

What else is running on the workstation?
If you have other software, including a firewall, try switching it off to see if the problem persists.

Is the problem reproducible on the same workstation?
Knowing that this defect occurs every time the described steps are taken is helpful because it indicates a straightforward programming error. If the problem occurs at alternate times, or occasionally, thread interaction and timing problems in general are much more likely.

Is the problem reproducible on another workstation?
A problem that is not evident on another workstation might help you find the cause. A difference in hardware might make the problem disappear; for example, the number of processors. Also, differences in the operating system and application software installed might make a difference to the JVM. For example, the visibility of a race condition in the JVM or a user Java application might be influenced by the speed at which certain operations are performed by the system.

Does the problem occur on multiple platforms?
If the problem occurs only on one platform, it might be related to a platform-specific part of the JVM. Alternatively, it might be related to local code used inside a user application. If the problem occurs on multiple platforms, the problem might be related to the user Java application. Alternatively, it might be related to a cross-platform part of the JVM such as the Java Swing API. Some problems might be evident only on particular hardware; for example, Intel 32 bit architecture. A problem on particular hardware might indicate a JIT problem.

Can you reproduce the problem with the latest Service Refresh?
The problem might also have been fixed in a recent service refresh. Make sure

that you are using the latest service refresh for your environment. Check the latest details on the product Web site http://www-01.ibm.com/software/webservers/realtime/ or on http://www.ibm.com/developerWorks.

Are you using a supported Operating System (OS) with the latest patches installed?

It is important to use a supported operating system with the latest patches applied. For example, upgrading system libraries can solve problems. Later versions of system software can provide a richer set of diagnostic information. For more information, see "Problem determination" on page 125 . Check for latest details on http://www.ibm.com/developerworks.

Does turning off the JIT or AOT help?

If turning off the JIT or AOT prevents the problem, there might be a problem with the JIT or AOT. The problem can also indicate a race condition in your Java application that surfaces only in certain conditions. If the problem is intermittent, reducing the JIT compilation threshold to 0 might help reproduce the problem more consistently. (See "JIT and AOT problem determination" on page 241.)

Have you tried reinstalling the JVM or other software and rebuilding relevant application files?

Some problems occur from a damaged or incorrect installation of the JVM or other software. It is also possible that an application might have inconsistent versions of binary files or packages. Inconsistency is likely in a development or testing environment and could potentially be solved by getting a fresh build or installation.

Is the problem particular to a multiprocessor (or SMP) platform? If you are working on a multiprocessor platform, does the problem still exist on a uniprocessor platform?

This information is valuable to IBM Service.

Have you installed the latest patches for other software that interacts with the JVM? For example, the IBM WebSphere Application Server and DB2®.

The problem might be related to configuration of the JVM in a larger environment, and might have been solved already in a fix pack. Is the problem reproducible when the latest patches have been installed?

Have you enabled core dumps?

Core dumps are essential to enable IBM Service to debug a problem. Core dumps are enabled by default for the Java process. See "Using dump agents" on page 159 for details. The operating system settings might also need to be in place to enable the dump to be generated and to ensure that it is complete. Details of the required settings are contained in "Problem determination" on page 125 .

Are you using shared class caches?

Ensure that the name of the cache does not exceed 53 characters.

What logging information is available?

Information about any problems is produced by the JVM. You can enable more detailed logging, and control where the logging information goes.

Problem determination

This section describes problem determination for WebSphere Real Time for Real Time Linux.

The topics are:

Use the `man` command to obtain reference information about many of the commands mentioned in this set of topics.

Setting up and checking your Linux environment

Linux operating systems undergo a large number of patches and updates.

IBM personnel cannot test the JVM against every patch. The intention is to test against the most recent releases of a few distributions. In general, you should keep systems up-to-date with the latest patches.

The Java service team has a tool named ReportEnv that plugs into your JVM and reports on the JVM environment in real time. Your JVM environment affects the operation of the JVM. ReportEnv reports on environment variables and command-line parameters. It is a GUI tool, although it can be run without a GUI. The GUI allows you to browse your environment and, to some extent, dynamically change it. The tool also has a mechanism to generate reports to tell you the exact state of your JVM environment. The ReportEnv tool is available on request from jvmcookbook@uk.ibm.com.

Working directory

The current working directory of the JVM process is the default location for the generation of core files, Java dumps, heap dumps, and the JVM trace outputs, including Application Trace and Method trace. Enough free disk space must be available for this directory. Also, the JVM must have write permission.

Linux system dumps (core files)

When a crash occurs, the most important diagnostic data to obtain is the system dump. To ensure that this file is generated, you must check the following settings.

Operating system settings

Operating system settings must be correct. These settings can vary by distribution and Linux version.

To obtain full core files, set the following ulimit options:

```
ulimit -c unlimited   turn on corefiles with unlimited size
ulimit -n unlimited   allows an unlimited number of open file descriptors
ulimit -m unlimited   sets the user memory limit to unlimited
ulimit -f unlimited   sets the file size to unlimited
```

The current ulimit settings can be displayed using:

```
ulimit -a
```

These values are the "soft" limit, and are set for each user. These values cannot exceed the "hard" limit value. To display and change the "hard" limits, the same ulimit commands can be run using the additional **-H** flag. From Java 5, the `ulimit -c` value for the soft limit is ignored and the hard limit value is used to help ensure generation of the core file. You can disable core file generation by using the **-Xdump:system:none** command-line option.

Java Virtual Machine settings

To generate core files when a crash occurs, check that the JVM is set to do so.

Run `java -Xrealtime -Xdump:what`, which should produce the following:

```
-Xdump:system:
    events=gpf+abort,
    label=/mysdk/sdk/jre/bin/core.%Y%m%d.%H%M%S.%pid.dmp,
    range=1..0,
    priority=999,
    request=serial
```

The values above are the default settings. At least `events=gpf` must be set to generate a core file when a crash occurs. You can change and set options with the command-line option
-Xdump:system[:name1=value1,name2=value2 ...]

Available disk space

The available disk space must be large enough for the core file to be written.

The JVM allows the core file to be written to any directory that is specified in the `label` option. For example:

```
-Xdump:system:label=/mysdk/sdk/jre/bin/core.%Y%m%d.%H%M%S.%pid.dmp
```

To write the core file to this location, disk space must be sufficient (up to 4 GB might be required for a 32-bit process), and the correct permissions for the Java process to write to that location.

ZipException or IOException on Linux

When using a large number of file descriptors to load different instances of classes, you might see an error message "`java.util.zip.ZipException: error in opening zip file`", or some other form of IOException advising that a file could not be opened. The solution is to increase the provision for file descriptors, using the ulimit command. To find the current limit for open files, use the command:

```
ulimit -a
```

To allow more open files, use the command:

```
ulimit -n 8196
```

Using CPU Time limits to control runaway tasks

Because real time threads run at high priorities and with FIFO scheduling, failing applications (typically with tight CPU-bound loops) can cause a system to become unresponsive. In a development environment it can be useful to ensure runaway tasks are killed by limiting the amount of CPU that tasks might consume. See "Linux system dumps (core files)" on page 125 for a discussion on soft and hard limit settings.

The command `ulimit -t` lists the current timeout value in CPU seconds. This value can be reduced with either soft, for example, `ulimit -St 900` to set the soft timeout to 15 minutes or hard values to stop runaway tasks.

Related concepts

"Hardware and software prerequisites" on page 9
Use this list to check the hardware, operating system, and Java environment that is supported for WebSphere Real Time for RT Linux.

Related information

"Installing a Real Time Linux environment" on page 10
Before you can install WebSphere Real Time for RT Linux, you must install Real Time Linux.

General debugging techniques

This section provides a guide to the JVM-provided diagnostic tools and Linux commands that can be useful when you are diagnosing problems that occur with the Linux JVM.

Action	Reference
Starting Javadumps	See "Using Javadump" on page 174.
Starting Heapumps	See "Using Heapdump" on page 187.

Using the dump extractor

When a system (core) dump occurs, you must use the dump extractor to prepare the dump for analysis.

To use the dump extractor, you need:

- The system dump (core file)
- A copy of the Java executable that was running the process
- Copies of all the libraries that were in use when the system dump was created

When a system dump is generated, run the jextract utility against the system dump:

```
jextract -Xrealtime <system dump name>
```

to generate a file called `dumpfilename.zip` in the current directory. `dumpfilename.zip` is a compressed file containing the required files. Running jextract against the system dump also allows for the subsequent use of the dump viewer.

See "Using system dumps and the dump viewer" on page 192 for more information.

Using system dump tools

The commands objdump and nm are used to investigate and display information about system (core) dumps. If a crash occurs and a system dump is produced, these commands help you analyze the file.

About this task

Run these commands on the same workstation as the one that produced the system dumps to use the most accurate symbol information available. This output (together with the system dump, if small enough) is used by the IBM support team for Java to diagnose a problem.

objdump

Use this command to disassemble shared objects and libraries. After you have discovered which library or object has caused the problem, use objdump to locate the method in which the problem originates. To start objdump, enter: `objdump <option> <filename>`

You can see a complete list of options by typing `objdump -H`. The **-d** option disassembles contents of executable sections

nm This command lists symbol names from object files. These symbol names can be either functions, global variables, or static variables. For each symbol, the value, symbol type, and symbol name are displayed. Lower case symbol types mean the symbol is local, while upper case means the symbol is global or external. To use this tool, type: `nm <option> <system dump>`.

Examining process information

The kernel provides useful process and environment information. These commands can be used to view this information.

The ps command

On Linux, Java threads are implemented as system threads and might be visible in the process table, depending on the Linux distribution.

Running the ps command gives you a snapshot of the current processes. The ps command gets its information from the `/proc` file system. From WebSphere Real Time for RT Linux V2 SR3, Java thread names are visible in the operating system, although the full thread name might be truncated. Here is an example of using ps:

```
ps -eLo pid,tid,rtprio,comm,cmd
29286 29286      - java            jre/bin/java -Xrealtime -jar example.jar
29286 29287      - main            jre/bin/java -Xrealtime -jar example.jar
29286 29290     88 Signal Reporter jre/bin/java -Xrealtime -jar example.jar
29286 29295      - JIT Compilation jre/bin/java -Xrealtime -jar example.jar
29286 29296     13 JIT Sampler     jre/bin/java -Xrealtime -jar example.jar
29286 29297      - Signal Dispatch jre/bin/java -Xrealtime -jar example.jar
29286 29298      - Finalizer maste jre/bin/java -Xrealtime -jar example.jar
29286 29299     11 Gc Slave Thread jre/bin/java -Xrealtime -jar example.jar
29286 29300     89 Metronome GC Al jre/bin/java -Xrealtime -jar example.jar
29286 29301      - Thread-2        jre/bin/java -Xrealtime -jar example.jar
29286 29302     43 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29303     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29304     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29305     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29306     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29307     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29311     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29312     83 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
```

```
29286 29313     85 Realtime AEH No jre/bin/java -Xrealtime -jar example.jar
29286 29314     85 Realtime AEH No jre/bin/java -Xrealtime -jar example.jar
29286 29315     87 Realtime Schedu jre/bin/java -Xrealtime -jar example.jar
29286 29316     79 Realtime AEH Se jre/bin/java -Xrealtime -jar example.jar
29286 29317     85 Realtime Non-he jre/bin/java -Xrealtime -jar example.jar
29286 29318     83 Realtime Heap T jre/bin/java -Xrealtime -jar example.jar
29286 29319     83 Realtime Heap T jre/bin/java -Xrealtime -jar example.jar
29286 29321     45 RealtimeThread- jre/bin/java -Xrealtime -jar example.jar
29286 29343     43 RealtimeThread- jre/bin/java -Xrealtime -jar example.jar
29286 29345      - stdout reader j jre/bin/java -Xrealtime -jar example.jar
29286 29346      - stderr reader j jre/bin/java -Xrealtime -jar example.jar
```

e Selects all processes.

L Shows threads.

o Provides a pre-defined format of columns to display. The columns specified are the process ID, thread ID, scheduling policy, real-time thread priority, and the command associated with the process. This information is useful for understanding what threads in your application as well as the virtual machine are running at a given time.

The top command

The top command displays the most CPU-intensive or memory-intensive processes in real time. It provides an interactive interface for manipulation of processes and allows sorting by different criteria, such as CPU usage or memory usage. Press **h** while running top to see all the available interactive commands.

The top command displays several fields of information for each process. The process field shows the total number of processes that are running, but breaks down the information into tasks that are running, sleeping, stopped, or undead. In addition to displaying PID, PPID, and UID, the top command displays information about memory usage and swap space. The `mem` field shows statistics on memory usage, including available memory, free memory, used memory, shared memory, and memory used for buffers. The swap field shows total swap space, available swap space, and used swap space.

The vmstat command

The vmstat command reports virtual storage statistics. It is useful to perform a general health check on your system because it reports on the system as a whole. Commands such as ps and top can be used to gain more specific information about the process operation.

When you use it for the first time during a session, the information is reported as averages since the last reboot. Further usage produces reports that are based on a sampling period that you can specify as an option. `vmstat 3 4` displays values every 3 seconds for a count of four times. It might be useful to start vmstat before the application, have it direct its output to a file and later study the statistics as the application started and ran.

The basic output from this command is displayed in these sections:

processes
Shows how many processes are awaiting run time, blocked, or swapped out.

memory
Shows the amount of memory (in kilobytes) swapped, free, buffered, and

cached. If the free memory is going down during certain stages of your applications execution, there might be a memory leak.

swap Shows the kilobytes per second of memory swapped in from and swapped out to disk. Memory is swapped out to disk if not enough RAM is available to store it all. Large values here can be a hint that not enough RAM is available (although it is normal to get swapping when the application first starts).

io Shows the number of blocks per second of memory sent to and received from block devices.

system
Displays the interrupts and the context switches per second. There is a performance penalty associated with each context switch so a high value for this section might mean that the program does not scale well.

cpu Shows a breakdown of processor time between user time, system time, and idle time. The idle time figure shows how busy a processor is, with a low value indicating that the processor is busy. You can use this knowledge to help you understand which areas of your program are using the CPU the most.

ldd

The Linux command **ldd** prints information that should help you to work out the shared library dependency of your application.

Tracing tools

Tracing is a technique that presents details of the execution of your program. If you are able to follow the path of execution, you will gain a better insight into how your program runs and interacts with its environment.

Also, you will be able to pinpoint locations where your program starts to deviate from its expected behavior.

Three tracing tools on Linux are strace, ltrace, and mtrace. The command `man strace` displays a full set of available options.

strace
The strace tool traces system calls. You can either use it on a process that is already available, or start it with a new process. strace records the system calls made by a program and the signals received by a process. For each system call, the name, arguments, and return value are used. strace allows you to trace a program without requiring the source (no recompilation is required). If you use strace with the **-f** option, it will trace child processes that have been created as a result of a forked system call. You can use strace to investigate plug-in problems or to try to understand why programs do not start properly.

To use strace with a Java application, type `strace java -Xrealtime <class-name>`.

You can direct the trace output from the strace tool to a file by using the `-o` option.

ltrace
The ltrace tool is distribution-dependent. It is very similar to strace. This tool intercepts and records the dynamic library calls as called by the executing process. strace does the same for the signals received by the executing process.

To use ltrace with a Java application, type `ltrace java -Xrealtime <class-name>`

mtrace

mtrace is included in the GNU toolset. It installs special handlers for malloc, realloc, and free, and enables all uses of these functions to be traced and recorded to a file. This tracing decreases program efficiency and should not be enabled during normal use. To use mtrace, set **IBM_MALLOCTRACE** to 1, and set **MALLOC_TRACE** to point to a valid file where the tracing information will be stored. You must have write access to this file.

To use mtrace with a Java application, type:

```
export IBM_MALLOCTRACE=1
export MALLOC_TRACE=/tmp/file
java -Xrealtime <class-name>
mtrace /tmp/file
```

Debugging with gdb

The GNU debugger (gdb) allows you to examine the internals of another program while the program executes or retrospectively to see what a program was doing at the moment that it crashed.

The gdb allows you to examine and control the execution of code and is useful for evaluating the causes of crashes or general incorrect behavior. gdb does not handle Java processes, so it is of limited use on a pure Java program. It is useful for debugging native libraries and the JVM itself.

Running gdb

You can run gdb in three ways:

Starting a program

Typically the command: `gdb <application>` is used to start a program under the control of gdb. However, because of the way that Java is launched, you must start gdb by setting an environment variable and then calling Java:

```
export IBM_JVM_DEBUG_PROG=gdb
java
```

Then you receive a gdb prompt, and you supply the run command and the Java arguments:

```
r <java_arguments>
```

Attaching to a running program

If a Java program is already running, you can control it under gdb. The process ID of the running program is required, and then gdb is started with the Java application as the first argument and the process ID as the second argument:

```
gdb <Java Executable> <PID>
```

When gdb is attached to a running program, this program is halted and its position in the code is displayed for the viewer. The program is then under the control of gdb and you can start to issue commands to set and view the variables and generally control the execution of the code.

Running on a system dump (corefile)

A system dump is typically produced when a program crashes. gdb can be run on this system dump. The system dump contains the state of the program when the crash occurred. Use gdb to examine the values of all the variables and registers leading up to a crash. This information helps you discover what caused the crash. To debug a system dump, start gdb with the Java application file as the first argument and the system dump name as the second argument:

```
gdb <Java Executable> <system dump>
```

When you run gdb against a system dump, it initially shows information such as the termination signal the program received, the function that was executing at the time, and even the line of code that generated the fault.

When a program comes under the control of gdb, a welcome message is displayed followed by a prompt (gdb). The program is now waiting for you to enter instructions. For each instruction, the program continues in whichever way you choose.

Setting breakpoints and watchpoints

Breakpoints can be set for a particular line or function using the command:

```
break linenumber
```

or

```
break functionName
```

After you have set a breakpoint, use the `continue` command to allow the program to execute until it reaches a breakpoint.

Set breakpoints using conditionals so that the program halts only when the specified condition is reached. For example, using `breakpoint 39 if var == value` causes the program to halt when it reaches line 39, but only if the variable is equal to the specified value.

If you want to know *where* as well as *when* a variable became a certain value you can use a watchpoint. Set the watchpoint when the variable in question is in scope. After doing so, you will be alerted whenever this variable attains the specified value. The syntax of the command is: `watch var == value`.

To see which breakpoints and watchpoints are set, use the `info` command:

```
info break
info watch
```

When gdb reaches a breakpoint or watchpoint, it prints out the line of code it is next set to execute. Setting a breakpoint at line 8 will cause the program to halt after completing execution of line 7 but before execution of line 8. As well as breakpoints and watchpoints, the program also halts when it receives certain system signals. By using the following commands, you can stop the debugging tool halting every time it receives these system signals:

```
handle sig32 pass nostop noprint
handle sigusr2 pass nostop noprint
```

Examining the code

When the correct position of the code has been reached, there are a number of ways to examine the code. The most useful is `backtrace` (abbreviated to `bt`), which shows the call stack. The call stack is the collection of function frames, where each function frame contains information such as function parameters and local variables. These function frames are placed on the call stack in the order that they are executed. This means that the most recently called function is displayed at the top of the call stack. You can follow the trail of execution of a program by examining the call stack. When the call stack is displayed, it shows a frame

number on the left side, followed by the address of the calling function, followed by the function name and the source file for the function. For example:

```
#6 0x804c4d8 in myFunction () at myApplication.c
```

To view more detailed information about a function frame, use the `frame` command along with a parameter specifying the frame number. After you have selected a frame, you can display its variables using the command `print var`.

Use the `print` command to change the value of a variable; for example, `print var = newValue`.

The `info locals` command displays the values of all local variables in the selected function.

To follow the exact sequence of execution of your program, use the `step` and `next` commands. Both commands take an optional parameter specifying the number of lines to execute. However, `next` treats function calls as a single line of execution, while `step` progresses through each line of the called function, one step at a time.

Useful commands

When you have finished debugging your code, the `run` command causes the program to run through to its end or its crash point. The `quit` command is used to exit gdb.

Other useful commands are:

`ptype`
Prints data type of variable.

`info share`
Prints the names of the shared libraries that are currently loaded.

`info functions`
Prints all the function prototypes.

`list`
Shows the 10 lines of source code around the current line.

`help`
Displays a list of subjects, each of which can have the help command called on it, to display detailed help on that topic.

Related information

"Using system dumps and the dump viewer" on page 192
The JVM can generate native system dumps, also known as core dumps, under configurable conditions. System dumps are typically quite large. Use the gdb tool to analyze a system dump on Linux.

Diagnosing crashes

Many approaches are possible when you are trying to determine the cause of a crash. The process typically involves isolating the problem by checking the system setup and trying various diagnostic options.

Checking the system environment

The system might have been in a state that has caused the JVM to crash. For example, this could be a resource shortage (such as memory or disk) or a stability

problem. Check the Javadump file, which contains various system information (as described in "Using Javadump" on page 174). The Javadump file tells you how to find disk and memory resource information. The system logs can give indications of system problems.

Gathering process information

It is useful to find out what exactly was happening leading up to the crash.

Use gdb and the bt command to display the stack trace of the failing thread and show what was running up to the point of the crash. This could be:

- JNI native code.
- JIT or AOT compiled code. If you have a problem with JIT or AOT code, try running without the JIT or AOT code by using the **-Xint** option.
- JVM code.

Other tracing methods:

- ltrace
- strace
- mtrace - can be used to track memory calls and determine possible corruption
- RAS trace, described in *Using the Reliability, Availability, and Servicability Interface* in the Diagnostics Guide.

Finding out about the Java environment

Use the Javadump to determine what each thread was doing and which Java methods were being executed. Match function addresses against library addresses to determine the source of code executing at various points.

Use the **-verbose:gc** option to look at the state of the Java heap and the Immortal and Scoped Memory areas, and determine if:

- There was a shortage of memory in one of the memory areas and if this could have caused the crash.
- The crash occurred during garbage collection, indicating a possible garbage collection fault.
- The crash occurred after garbage collection, indicating a possible memory corruption.

Debugging hangs

A hang is caused by a wait (also known as a deadlock) or a loop (also known as a livelock). A deadlock sometimes occurs because of a wait on a lock or monitor. A loop can occur similarly or sometimes because of an algorithm making little or no progress towards completion.

A wait could either be caused by a timing error leading to a missed notification, or by two threads deadlocking on resources.

For an explanation of deadlocks and diagnosing them using a Javadump, see "Locks, monitors, and deadlocks (LOCKS)" on page 180.

A loop is caused by a thread failing to exit a loop in a timely manner. This might be because it calculated the wrong limit value or missed a flag that was intended to exit the loop. This problem might only occur on multi-processor workstations

and if this is the case it can usually be traced to a failure to make the flag volatile or access it whilst holding an appropriate monitor.

The following approaches are useful to resolve waits and loops:

- Monitoring process and system state (as described in "MustGather information for Linux" on page 138).
- Javadumps give monitor and lock information. You can trigger a Javadump during a hang by using the `kill -QUIT <PID>` command.

Debugging memory leaks

If dynamically allocated objects are not freed at the end of their lifetime, memory leaks can occur. When objects that should have had their memory released are still holding memory and more objects are being created, the system eventually runs out of memory.

The **mtrace** tool from GNU is available for tracking memory calls. This tool enables you to trace memory calls such as malloc and realloc so that you can detect and locate memory leaks.

For more details about analyzing the Java Heap, see "Using Heapdump" on page 187.

Debugging performance problems

Locating the causes of poor performance is often difficult. Although many factors can affect performance, the overall effect is generally perceived as poor response or slow execution of your program.

Correcting one performance problem might cause more problems in another area. By finding and correcting a bottleneck in one place you might only shift the cause of poor performance to other areas. To improve performance, experiment with tuning different parameters, monitoring the effect, and retuning until you are satisfied that your system is performing acceptably.

Finding the bottleneck

Given that any performance problem could be caused by any one of several factors, you must look at several areas to eliminate each one.

Determine which resource is constraining the system:

- CPU
- Memory
- Input/Output (I/O)

Several tools are available that enable you to measure system components and establish how they are performing and under what kind of workload.

The key things to measure are CPU usage and memory usage. If the CPU is not powerful enough to handle the workload, it will be impossible to tune the system to make much difference to overall performance. You must upgrade the CPU. Similarly, if a program is running in an environment without enough memory, an increase in the memory improves performance far more than any amount of tuning.

CPU usage

Java processes consume 100% of processor time when they reach their resource limits. Ensure that ulimit settings are appropriate to the application requirement.

See “Linux system dumps (core files)” on page 125 for more information about ulimit.

The `/proc` file system provides information about all the processes that are running on your system, including the Linux kernel. See `man proc` from a Linux shell for official Linux documentation about the `/proc` file system.

The top command provides real-time information about your system processes. The top command is useful for getting an overview of the system load. It clearly displays which processes are using the most resources. Having identified the processes that are probably causing a degraded performance, you can take further steps to improve the overall efficiency of your program. More information is provided about the top command in “The top command” on page 129.

Memory usage

If a system is performing poorly because of lack of memory resources, it is memory bound. By viewing the contents of `/proc/meminfo`, you can view your memory resources and see how they are being used. `/proc/swap` contains information on your swap file.

Swap space is used as an extension of the systems virtual storage. Therefore, not having enough memory or swap space causes performance problems. A general guideline is that swap space should be at least twice as large as the physical memory.

A swap space can be either a file or disk partition. A disk partition offers better performance than a file does. fdisk and cfdisk are the commands that you use to create another swap partition. It is a good idea to create swap partitions on different disk drives because this distributes the I/O activities and thus reduces the chance of further bottlenecks.

The vmstat tool helps you find where performance problems might be caused. For example, if you see that high swap rates are occurring, you probably do not have enough physical or swap space. The free command displays your memory configuration; `swapon -s` displays your swap device configuration. A high swap rate (for example, many page faults) means that you probably need to increase your physical memory. More information about the vmstat command are provided in “The vmstat command” on page 129.

Network problems

Another area that often affects performance is the network. Obviously, the more you know about the behavior of your program, the easier it is for you to decide whether this is a likely source of performance bottleneck.

If you think that your program is likely to be network I/O bound, netstat is a useful tool. In addition to providing information about network routes, netstat gives a list of active sockets for each network protocol and can give overall statistics, such as the number of packets that are received and sent.

Using netstat, you can see how many sockets are in a CLOSE_WAIT or ESTABLISHED state and you can tune the TCP/IP parameters accordingly for

better performance of the system. For example, tuning `/proc/sys/net/ipv4/tcp_keepalive_time` will reduce the time for socket waits in TIMED_WAIT state before closing a socket.

If you are tuning the `/proc/sys/net` file system, the effect will be on all the applications running on the system. To make a change to an individual socket or connection, use Java Socket API calls (on the appropriate socket object). Use `netstat -p` (or the lsof command) to find the PID of the process that owns a particular socket and its stack trace from a javacore file taken with the `kill -QUIT <pid>` command.

You can also use IBM's RAS trace, **-Xtrace:print=net**, to trace out network-related activity in the JVM. This technique is helpful when socket-related Java thread hangs are seen. Correlating output from `netstat -p`, lsof, JVM net trace, and `ps -efH` can help you to diagnose the network-related problems.

Providing summary statistics that are related to your network is useful for investigating programs that might be under-performing because of TCP/IP problems. The more you understand your hardware capacity, the easier it is for you to tune with confidence the parameters of particular system components that will improve the overall performance of your application. You can also determine whether tuning the system noticeably improves performance or whether you require system upgrades.

Sizing memory areas

The JVM can be tuned by varying the sizes of the heap, immortal and scoped memory. Choose the correct size to optimize performance. Using the correct size can make it easier for the Garbage Collector to provide the required utilization.

For more information about varying the size of the memory areas see "Troubleshooting the Metronome Garbage Collector" on page 51.

JIT compilation and performance

The JIT is another area that can affect the performance of your program. When deciding whether or not to use JIT compilation, you must make a balance between faster execution and increased processor usage during compilation. When using the JIT, you should consider the implications to real-time behavior.

If you require predictable behavior but also need better performance then you should consider using ahead-of-time (AOT) compilation. For further information see Chapter 4, "Using compiled code with WebSphere Real Time for RT Linux," on page 23.

Related information

"JIT and AOT problem determination" on page 241
You can use command-line options to help diagnose JIT and AOT compiler problems and to tune performance.

Application profiling

You can learn a lot about your Java application by using the hprof profiling agent. Statistics about CPU and memory usage are presented along with many other options.

The hprof tool is discussed in detail in

The **-Xrunhprof:help** command-line option displays a list of suboptions that you can use with hprof.

The Performance Inspector package contains a suite of performance analysis tools for Linux. You can use tools to help identify performance problems in your application as well as to understand how your application interacts with the Linux kernel. See http://perfinsp.sourceforge.net/ for details.

MustGather information for Linux

When a problem occurs, the more information known about the state of the system environment, the easier it is to reach a diagnosis of the problem.

A large set of information can be collected, although only some of it will be relevant for particular problems. The following sections tell you the data to collect to help the IBM service team for Java solve the problem.

Collecting system dumps (core files)

Collect system dumps to help diagnose many types of problem. Process the system dump with jextract. The resultant xml file is useful for service (see "Using the dump viewer" on page 194).

Producing system dumps

You can use the **-Xdump:system** command line option to obtain system dumps based on a trigger. See "Using dump agents" on page 159 for more information.

You can also use a Linux system utility to generate system dumps:

1. Determine the Process ID of your application using the ps command. See "The ps command" on page 128.
2. At a shell prompt, type `gcore -o <dump file name> <pid>`

A system dump file is produced for your application. The application will be suspended while the system dump is written.

Process the system dump with jextract. The resultant jar file is useful for service (see "Using the dump viewer" on page 194).

Producing Javadumps

In some conditions, a crash, for example, a Javadump is produced, usually in the current directory.

In others for example, a hang, you might have to prompt the JVM for this by sending the JVM a SIGQUIT symbol:

1. Determine the Process ID of your application using the ps command. See "The ps command" on page 128.
2. At a shell prompt, type `kill -QUIT <pid>`

This is discussed in more detail in "Using Javadump" on page 174.

Producing Heapdumps

The JVM can generate a Heapdump at the request of the user, for example by calling com.ibm.jvm.Dump.HeapDump() from inside the application, or by default when the JVM terminates because of an OutOfMemoryError. You can specify finer control of the timing of a Heapdump with the **-Xdump:heap** option. For example,

you could request a heapdump after a certain number of full garbage collections have occurred. The default heapdump format (phd files) is not human-readable and you process it using available tools such as Heaproots.

Producing Snap traces

Under default conditions, a running JVM collects a small amount of trace data in a special wraparound buffer. This data is dumped to file when the JVM terminates unexpectedly or an OutOfMemoryError occurs. You can use the **-Xdump:snap** option to vary the events that cause a snap trace to be produced. The snap trace is in normal trace file format and requires the use of the supplied standard trace formatter so that you can read it. See "Snap traces" on page 165 for more information about the contents and control of snap traces.

Using system logs

The kernel logs system messages and warnings. The system log is located in the `/var/log/messages` file. Use it to observe the actions that led to a particular problem or event. The system log can also help you determine the state of a system. Other system logs are in the `/var/log` directory.

Determining the operating environment

This section looks at the commands that can be useful to determine the operating environment of a process at various stages of its life-cycle.

uname -a
Displays operating system and hardware information.

df Displays free disk space on a system.

free
Displays memory use information.

ps -eLo pid,tid,policy,rtprio,comm,command
Displays a full process list.

lsof
Displays open file handles.

top
Displays process information (such as processor, memory, states) sorted by default by processor usage.

vmstat
Displays general memory and paging information.

The uname, df, and free output is the most useful. The other commands can be run before and after a crash or during a hang to determine the state of a process and to provide useful diagnostic information.

Sending information to Java Support

When you have collected the output of the commands listed in the previous section, put that output into files.

Compress the files (which could be very large) before sending them to Java Support. You should compress the files at a very high ratio.

The following command builds an archive from files {file1,..,fileN} and compresses them to a file with a name in the format filename.tgz:

```
tar czf filename.tgz file1 file2...filen
```

Collecting additional diagnostic data

Depending on the type of problem, the following data can also help you diagnose problems. The information available depends on the way in which Java is started and also the system environment. You will probably have to change the setup and then restart Java to reproduce the problem with these debugging aids switched on.

`/proc` file system

The `/proc` file system gives direct access to kernel level information. The `/proc/<pid>` directory contains detailed diagnostic information about the process with PID (process id) <*pid*>, where <*pid*> is the id of the process.

The command `cat /proc/<pid>/maps` lists memory segments (including native heap) for a given process.

strace, ltrace, and mtrace

Use the commands strace, ltrace, and mtrace to collect further diagnostic data. See "Tracing tools" on page 130.

Known limitations on Linux

Linux has been under rapid development and there have been various issues with the interaction of the JVM and the operating system, particularly in the area of threads.

Note the following limitations that might be affecting your Linux system.

Threads as processes

If the number of Java threads exceeds the maximum number of processes allowed, your program might:

- Get an error message
- Get a **SIGSEGV** error
- Stop

For more information, see *The Volano Report* at http://www.volano.com/report/index.html.

Floating stacks limitations

If you are running without floating stacks, regardless of what is set for **-Xss**, a minimum native stack size of 256 KB for each thread is provided.

On a floating stack Linux system, the **-Xss** values are used. If you are migrating from a non-floating stack Linux system, ensure that any **-Xss** values are large enough and are not relying on a minimum of 256 KB.

glibc limitations

If you receive a message indicating that the `libjava.so` library could not be loaded because of a symbol not found (such as __bzero), you might have an earlier version of the GNU C Runtime Library, glibc, installed. The SDK for Linux thread implementation requires glibc version 2.3.2 or greater.

Font limitations

When you are installing on a Red Hat system, to allow the font server to find the Java TrueType fonts, run (on Linux IA32, for example):

```
/usr/sbin/chkfontpath --add /opt/ibm/ibm-wrt-i386-60/jre/lib/fonts
```

You must do this at installation time and you must be logged on as "root" to run the command. For more detailed font issues, see the *Linux SDK and Runtime Environment User Guide*.

Performance issues on Linux Red Hat MRG kernels

A configuration issue with Red Hat MRG kernels can cause unexpected pauses to application threads when WebSphere Real Time starts with verbose garbage collection enabled. These pauses are not reported in the verbose GC output, but can last several milliseconds, depending on the network configuration. JVMs started from remotely defined LDAP users are affected the most, because the name service cache daemon (`nscd`) is not started, causing network delays. Solve the problem by starting `nscd`. Follow these steps to check on the status of the `nscd` service and correct the problem:

1. Check that the `nscd` daemon is running by typing the command:

   ```
   /sbin/service nscd status
   ```

   If the daemon is not running you see the following message:

   ```
   nscd is stopped
   ```

2. As root user, start the `nscd` service with the following command:

   ```
   /sbin/service nscd start
   ```

3. As root user, change the startup information for the `nscd` service with the following command:

   ```
   /sbin/chkconfig nscd on
   ```

The `nscd` process is now running, and starts automatically after reboot.

ORB problem determination

One of your first tasks when debugging an ORB problem is to determine whether the problem is in the client-side or in the server-side of the distributed application. Think of a typical RMI-IIOP session as a simple, synchronous communication between a client that is requesting access to an object, and a server that is providing it.

During this communication, a problem might occur in the execution of one of the following steps:

1. The client writes and sends a request to the server.
2. The server receives and reads the request.
3. The server executes the task in the request.
4. The server writes and sends a reply back.

5. The client receives and reads the reply.

It is not always easy to identify where the problem occurred. Often, the information that the application returns, in the form of stack traces or error messages, is not enough for you to make a decision. Also, because the client and server communicate through their ORBs, if a problem occurs, both sides will probably record an exception or unusual behavior.

This section describes all the clues that you can use to find the source of the ORB problem. It also describes a few common problems that occur more frequently. The topics are:

Identifying an ORB problem

A background of the constituents of the IBM ORB component.

What the ORB component contains

The ORB component contains the following:

- Java ORB from IBM and rmi-iiop runtime (com.ibm.rmi.*, com.ibm.CORBA.*)
- RMI-IIOP API (javax.rmi.CORBA.*,org.omg.CORBA.*)
- IDL to Java implementation (org.omg.* and IBM versions com.ibm.org.omg.*)
- Transient name server (com.ibm.CosNaming.*, org.omg.CosNaming.*) - tnameserv
- -iiop and -idl generators (com.ibm.tools.rmi.rmic.*) for the rmic compiler - rmic
- idlj compiler (com.ibm.idl.*)

What the ORB component does not contain

The ORB component does *not* contain:

- RMI-JRMP (also known as Standard RMI)
- JNDI and its plug-ins

Therefore, if the problem is in java.rmi.* or sun.rmi.*, it is not an ORB problem. Similarly, if the problem is in com.sun.jndi.*, it is not an ORB problem.

Platform dependent problems

If possible, run the test case on more than one platform. All the ORB code is shared. You can nearly always reproduce genuine ORB problems on any platform. If you have a platform-specific problem, it is likely to be in some other component.

JIT problem

JIT bugs are very difficult to find. They might show themselves as ORB problems. When you are debugging or testing an ORB application, it is always safer to switch off the JIT by setting the option **-Xint**.

Fragmentation

Disable fragmentation when you are debugging the ORB. Although fragmentation does not add complications to the ORB's functioning, a fragmentation bug can be difficult to detect because it will most likely show as a general marshalling problem. The way to disable fragmentation is to set the ORB property **com.ibm.CORBA.FragmentSize=0**. You must do this on the client side and on the server side.

ORB versions

The ORB component carries a few version properties that you can display by calling the main method of the following classes:

1. `com.ibm.CORBA.iiop.Version` (ORB runtime version)
2. com.ibm.tools.rmic.iiop.Version (for tools; for example, idlj and rmic)
3. rmic -iiop -version (run the command line for rmic)

Limitation with bidirectional GIOP

Bidirectional GIOP is not supported.

Debug properties

Properties to use to enable ORB traces.

Attention: Do not enable tracing for normal operation, because it might cause performance degradation. Even if you have switched off tracing, FFDC (First Failure Data Capture) is still working, so that only serious errors are reported. If a debug file is produced, examine it to check on the problem. For example, the server might have stopped without performing an ORB.shutdown().

You can use the following properties to enable the ORB traces:

- **com.ibm.CORBA.Debug:** This property turns on trace, message, or both. If you set this property to **trace**, only traces are enabled; if set to **message**, only messages are enabled. When set to **true**, both types are enabled; when set to **false**, both types are disabled. The default is **false**.
- **com.ibm.CORBA.Debug.Output:** This property redirects traces to a file, which is known as a trace log. When this property is not specified, or it is set to an empty string, the file name defaults to the format orbtrc.DDMMYYYY.HHmm.SS.txt, where D=Day; M=Month; Y=Year; H=Hour (24 hour format); m=Minutes; S=Seconds. Note that if the application (or Applet) does not have the privilege that it requires to write to a file, the trace entries go to stderr.
- **com.ibm.CORBA.CommTrace:** This property turns on wire tracing. Every incoming and outgoing GIOP message will be sent to the trace log. You can set this property independently from Debug; this property is useful if you want to look only at the flow of information, and you are not interested in debugging the internals. The only two values that this property can have are **true** and **false**. The default is **false**.

Here is an example of common usage:

```
java -Dcom.ibm.CORBA.Debug=true -Dcom.ibm.CORBA.Debug.Output=trace.log -Dcom.ibm.CORBA.CommTrace=true <classname>
```

For rmic -iiop or rmic -idl, the following diagnostic tools are available:

- **-J-Djavac.dump.stack=1:** This tool ensures that all exceptions are caught.
- **-Xtrace:** This tool traces the progress of the parse step.

If you are working with an IBM SDK, you can obtain CommTrace for the transient name server (tnameserv) by using the standard environment variable **IBM_JAVA_OPTIONS**. In a separate command session to the server or client SDKs, you can use:

```
export IBM_JAVA_OPTIONS=-Dcom.ibm.CORBA.CommTrace=true -Dcom.ibm.CORBA.Debug=true
```

The setting of this environment variable affects each Java process that is started, so use this variable carefully. Alternatively, you can use the **-J** option to pass the properties through the tnameserv wrapper, as follows:

```
tnameserv -J-Dcom.ibm.CORBA.Debug=true
```

ORB exceptions

The exceptions that can be thrown are split into user and system categories.

If your problem is related to the ORB, unless your application is doing nothing or giving you the wrong result, your log file or terminal is probably full of exceptions that include the words “CORBA” and “rmi” many times. All unusual behavior that occurs in a good application is highlighted by an exception. This principle also applies for the ORB with its CORBA exceptions. Similarly to Java, CORBA divides its exceptions into user exceptions and system exceptions.

User exceptions

User exceptions are IDL defined and inherit from org.omg.CORBA.UserException. These exceptions are mapped to checked exceptions in Java; that is, if a remote method raises one of them, the application that called that method must catch the exception. User exceptions are usually not fatal exceptions and should always be handled by the application. Therefore, if you get one of these user exceptions, you know where the problem is, because the application developer had to make allowance for such an exception to occur. In most of these cases, the ORB is not the source of the problem.

System exceptions

System exceptions are thrown transparently to the application and represent an unusual condition in which the ORB cannot recover gracefully, such as when a connection is dropped. The CORBA 2.6 specification defines 31 system exceptions and their mapping to Java. They all belong to the org.omg.CORBA package. The CORBA specification defines the meaning of these exceptions and describes the conditions in which they are thrown.

The most common system exceptions are:

- **BAD_OPERATION:** This exception is thrown when an object reference denotes an existing object, but the object does not support the operation that was called.

- **BAD_PARAM:** This exception is thrown when a parameter that is passed to a call is out of range or otherwise considered not valid. An ORB might raise this exception if null values or null pointers are passed to an operation.
- **COMM_FAILURE:** This exception is raised if communication is lost while an operation is in progress, after the request was sent by the client, but before the reply from the server has been returned to the client.
- **DATA_CONVERSION:** This exception is raised if an ORB cannot convert the marshaled representation of data into its native representation, or cannot convert the native representation of data into its marshaled representation. For example, this exception can be raised if wide character codeset conversion fails, or if an ORB cannot convert floating point values between different representations.
- **MARSHAL:** This exception indicates that the request or reply from the network is structurally not valid. This error typically indicates a bug in either the client-side or server-side runtime. For example, if a reply from the server indicates that the message contains 1000 bytes, but the actual message is shorter or longer than 1000 bytes, the ORB raises this exception.
- **NO_IMPLEMENT:** This exception indicates that although the operation that was called exists (it has an IDL definition), no implementation exists for that operation.
- **UNKNOWN:** This exception is raised if an implementation throws a non-CORBA exception, such as an exception that is specific to the implementation's programming language. It is also raised if the server returns a system exception that is unknown to the client. If the server uses a later version of CORBA than the version that the client is using, and new system exceptions have been added to the later version this exception can happen.

Completion status and minor codes

Two pieces of data are associated with each system exception, these are described in this section.

- A completion status, which is an enumerated type that has three values: COMPLETED_YES, COMPLETED_NO and COMPLETED_MAYBE. These values indicate either that the operation was executed in full, that the operation was not executed, or that the execution state cannot be determined.
- A long integer, called minor code, that can be set to some ORB vendor-specific value. CORBA also specifies the value of many minor codes.

Usually the completion status is not very useful. However, the minor code can be essential when the stack trace is missing. In many cases, the minor code identifies the exact location of the ORB code where the exception is thrown and can be used by the vendor's service team to localize the problem quickly. However, for standard CORBA minor codes, this is not always possible. For example:

```
org.omg.CORBA.OBJECT_NOT_EXIST: SERVANT_NOT_FOUND   minor code: 4942FC11  completed: No
```

Minor codes are usually expressed in hexadecimal notation (except for Sun's minor codes, which are in decimal notation) that represents four bytes. The OMG organization has assigned to each vendor a range of 4096 minor codes. The IBM vendor-specific minor code range is 0x4942F000 through 0x4942FFFF.

System exceptions might also contain a string that describes the exception and other useful information. You will see this string when you interpret the stack trace.

The ORB tends to map all Java exceptions to CORBA exceptions. A runtime exception is mapped to a CORBA system exception, while a checked exception is mapped to a CORBA user exception.

More exceptions other than the CORBA exceptions could be generated by the ORB component in a code bug. All the Java unchecked exceptions and errors and others that are related to the ORB tools rmic and idlj must be considered. In this case, the only way to determine whether the problem is in the ORB, is to look at the generated stack trace and see whether the objects involved belong to ORB packages.

Java security permissions for the ORB

When running with a Java SecurityManager, invocation of some methods in the CORBA API classes might cause permission checks to be made that could result in a SecurityException.

The following table shows methods affected when running with Java 2 SecurityManager:

Class/Interface	Method	Required permission
org.omg.CORBA.ORB	init	java.net.SocketPermission resolve
org.omg.CORBA.ORB	connect	java.net.SocketPermission listen
org.omg.CORBA.ORB	resolve_initial_references	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_is_a	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_non_existent	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	OutputStream _request (String, boolean)	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_get_interface_def	java.net.SocketPermission connect
org.omg.CORBA. Request	invoke	java.net.SocketPermission connect
org.omg.CORBA. Request	send_deferred	java.net.SocketPermission connect
org.omg.CORBA. Request	send_oneway	java.net.SocketPermission connect
javax.rmi. PortableRemoteObject	narrow	java.net.SocketPermission connect

If your program uses any of these methods, ensure that it is granted the necessary permissions.

Interpreting the stack trace

Whether the ORB is part of a middleware application or you are using a Java stand-alone application (or even an applet), you must retrieve the stack trace that is generated at the moment of failure. It could be in a log file, or in your terminal or browser window, and it could consist of several chunks of stack traces.

The following example describes a stack trace that was generated by a server ORB running in the WebSphere Application Server:

```
org.omg.CORBA.MARSHAL: com.ibm.ws.pmi.server.DataDescriptor; IllegalAccessException  minor code: 4942F23E  completed: No
         at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:199)
         at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
         at com.ibm.rmi.io.ValueHandlerImpl.read_Array(ValueHandlerImpl.java:625)
         at com.ibm.rmi.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:273)
         at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:189)
         at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
         at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie._invoke(_EJSRemoteStatelessPmiService_Tie.java:613)
         at com.ibm.CORBA.iiop.ExtendedServerDelegate.dispatch(ExtendedServerDelegate.java:515)
         at com.ibm.CORBA.iiop.ORB.process(ORB.java:2377)
         at com.ibm.CORBA.iiop.OrbWorker.run(OrbWorker.java:186)
         at com.ibm.ejs.oa.pool.ThreadPool$PooledWorker.run(ThreadPool.java:104)
         at com.ibm.ws.util.CachedThread.run(ThreadPool.java:137)
```

In the example, the ORB mapped a Java exception to a CORBA exception. This exception is sent back to the client later as part of a reply message. The client ORB reads this exception from the reply. It maps it to a Java exception (java.rmi.RemoteException according to the CORBA specification) and throws this new exception back to the client application.

Along this chain of events, often the original exception becomes hidden or lost, as does its stack trace. On early versions of the ORB (for example, 1.2.x, 1.3.0) the only way to get the original exception stack trace was to set some ORB debugging properties. Newer versions have built-in mechanisms by which all the nested stack traces are either recorded or copied around in a message string. When dealing with an old ORB release (1.3.0 and earlier), it is a good idea to test the problem on newer versions. Either the problem is not reproducible (known bug already solved) or the debugging information that you obtain is much more useful.

Description string

The example stack trace shows that the application has caught a CORBA org.omg.CORBA.MARSHAL system exception. After the MARSHAL exception, some extra information is provided in the form of a string. This string should specify minor code, completion status, and other information that is related to the problem. Because CORBA system exceptions are alarm bells for an unusual condition, they also hide inside what the real exception was.

Usually, the type of the exception is written in the message string of the CORBA exception. The trace shows that the application was reading a value (read_value()) when an IllegalAccessException occurred that was associated to class com.ibm.ws.pmi.server.DataDescriptor. This information is an indication of the real problem and should be investigated first.

Interpreting ORB traces

The ORB trace file contains messages, trace points, and wire tracing. This section describes the various types of trace.

Message trace

An example of a message trace.

Here is a simple example of a message:

```
19:12:36.306 com.ibm.rmi.util.Version logVersions:110 P=754534:O=0:CT
ORBRas[default] IBM Java ORB build orbdev-20050927
```

This message records the time, the package, and the method name that was called. In this case, logVersions() prints out, to the log file, the version of the running ORB.

After the first colon in the example message, the line number in the source code where that method invocation is done is written (110 in this case). Next follows the letter P that is associated with the process number that was running at that moment. This number is related (by a hash) to the time at which the ORB class was loaded in that process. It is unlikely that two different processes load their ORBs at the same time.

The following O=0 (alphabetic O = numeric 0) indicates that the current instance of the ORB is the first one (number 0). CT specifies that this is the main (control) thread. Other values are: LT for listener thread, RT for reader thread, and WT for worker thread.

The ORBRas field shows which RAS implementation the ORB is running. It is possible that when the ORB runs inside another application (such as a WebSphere application), the ORB RAS default code is replaced by an external implementation.

The remaining information is specific to the method that has been logged while executing. In this case, the method is a utility method that logs the version of the ORB.

This example of a possible message shows the logging of entry or exit point of methods, such as:

```
14:54:14.848 com.ibm.rmi.iiop.Connection <init>:504 LT=0:P=650241:O=0:port=1360 ORBRas[default] Entry
.....
14:54:14.857 com.ibm.rmi.iiop.Connection <init>:539 LT=0:P=650241:O=0:port=1360 ORBRas[default] Exit
```

In this case, the constructor (that is, <init>) of the class Connection is called. The tracing records when it started and when it finished. For operations that include the java.net package, the ORBRas logger prints also the number of the local port that was involved.

Comm traces

An example of comm (wire) tracing.

Here is an example of comm tracing:

```
 // Summary of the message containing name-value pairs for the principal fields
OUT GOING:
Request Message // It is an out going request, therefore we are dealing with a client
Date:           31 January 2003 16:17:34 GMT
Thread Info:    P=852270:O=0:CT
Local Port:     4899 (0x1323)
Local IP:       9.20.178.136
```

```
Remote Port:    4893 (0x131D)
Remote IP:      9.20.178.136
GIOP Version:   1.2
Byte order:     big endian

Fragment to follow: No     // This is the last fragment of the request
Message size:   276 (0x114)
--

Request ID:         5  // Request Ids are in ascending sequence
Response Flag:      WITH_TARGET // it means we are expecting a reply to this request
Target Address:     0
Object Key:         length = 26 (0x1A)  // the object key is created by the server when exporting
                                        // the servant and retrieved in the IOR using a naming service
                    4C4D4249 00000010 14F94CA4 00100000
                    00080000 00000000 0000
Operation:          message // That is the name of the method that the client invokes on the servant
Service Context:    length = 3 (0x3)   // There are three service contexts
Context ID:         1229081874 (0x49424D12) // Partner version service context. IBM only
Context data:       length = 8 (0x8)
                    00000000 14000005


Context ID:         1 (0x1)  // Codeset CORBA service context
Context data:       length = 12 (0xC)
                    00000000 00010001 00010100

Context ID:         6 (0x6)  // Codebase CORBA service context
Context data:       length = 168 (0xA8)
                    00000000 00000028 49444C3A 6F6D672E
                    6F72672F 53656E64 696E6743 6F6E7465
                    78742F43 6F646542 6173653A 312E3000
                    00000001 00000000 0000006C 00010200
                    0000000D 392E3230 2E313738 2E313336
                    00001324 0000001A 4C4D4249 00000010
                    15074A96 00100000 00080000 00000000
                    00000000 00000002 00000001 00000018
                    00000000 00010001 00000001 00010020
                    00010100 00000000 49424D0A 00000008
                    00000000 14000005
Data Offset:        11c
// raw data that goes in the wire in numbered rows of 16 bytes and the corresponding ASCII
decoding
0000: 47494F50 01020000 00000114 00000005    GIOP............
0010: 03000000 00000000 0000001A 4C4D4249    ............LMBI
0020: 00000010 14F94CA4 00100000 00080000    ......L.........
0030: 00000000 00000000 00000008 6D657373    ............mess
0040: 61676500 00000003 49424D12 00000008    age.....IBM.....
0050: 00000000 14000005 00000001 0000000C    ................
0060: 00000000 00010001 00010100 00000006    ................
0070: 000000A8 00000000 00000028 49444C3A    ...........(IDL:
0080: 6F6D672E 6F72672F 53656E64 696E6743    omg.org/SendingC
0090: 6F6E7465 78742F43 6F646542 6173653A    ontext/CodeBase:
00A0: 312E3000 00000001 00000000 0000006C    1.0............l
00B0: 00010200 0000000D 392E3230 2E313738    ........9.20.178
00C0: 2E313336 00001324 0000001A 4C4D4249    .136...$....LMBI
00D0: 00000010 15074A96 00100000 00080000    ......J.........
00E0: 00000000 00000000 00000002 00000001    ................
00F0: 00000018 00000000 00010001 00000001    ................
0100: 00010020 00010100 00000000 49424D0A    ... ........IBM.
0110: 00000008 00000000 14000005 00000000    ................
```

Note: The italic comments that start with a double slash have been added for clarity; they are not part of the traces.

In this example trace, you can see a summary of the principal fields that are contained in the message, followed by the message itself as it goes in the wire. In the summary are several field name-value pairs. Each number is in hexadecimal notation.

For details of the structure of a GIOP message, see the CORBA specification, chapters 13 and 15: http://www.omg.org/cgi-bin/doc?formal/99-10-07.

Client or server

From the first line of the summary of the message, you can identify whether the host to which this trace belongs is acting as a server or as a client. OUT GOING means that the message has been generated on the workstation where the trace was taken and is sent to the wire.

In a distributed-object application, a server is defined as the provider of the implementation of the remote object to which the client connects. In this work, however, the convention is that a client sends a request while the server sends back a reply. In this way, the same ORB can be client and server in different moments of the rmi-iiop session.

The trace shows that the message is an outgoing request. Therefore, this trace is a client trace, or at least part of the trace where the application acts as a client.

Time information and host names are reported in the header of the message.

The Request ID and the Operation ("message" in this case) fields can be very helpful when multiple threads and clients destroy the logical sequence of the traces.

The GIOP version field can be checked if different ORBs are deployed. If two different ORBs support different versions of GIOP, the ORB that is using the more recent version of GIOP should fall back to a common level. By checking that field, however, you can easily check whether the two ORBs speak the same language.

Service contexts

The header also records three service contexts, each consisting of a context ID and context data.

A service context is extra information that is attached to the message for purposes that can be vendor-specific such as the IBM Partner version.

Usually, a security implementation makes extensive use of these service contexts. Information about an access list, an authorization, encrypted IDs, and passwords could travel with the request inside a service context.

Some CORBA-defined service contexts are available. One of these is the Codeset.

In the example, the codeset context has `ID 1` and data `00000000 00010001 00010100`. Bytes 5 through 8 specify that characters that are used in the message are encoded in ASCII (00010001 is the code for ASCII). Bytes 9 through 12 instead are related to wide characters.

The default codeset is UTF8 as defined in the CORBA specification, although almost all Windows and UNIX® platforms typically communicate through ASCII. i5/OS® and Mainframes such as zSeries® systems are based on the IBM EBCDIC encoding.

The other CORBA service context, which is present in the example, is the Codebase service context. It stores information about how to call back to the client to access resources in the client such as stubs, and class implementations of parameter objects that are serialized with the request.

Common problems

This section describes some of the problems that you might find.

ORB application hangs

One of the worst conditions is when the client, or server, or both, hang. If a hang occurs, the most likely condition (and most difficult to solve) is a deadlock of threads. In this condition, it is important to know whether the workstation on which you are running has more than one CPU, and whether your CPU is using Simultaneous Multithreading (SMT).

A simple test that you can do is to keep only one CPU running, disable SMT, and see whether the problem disappears. If it does, you know that you must have a synchronization problem in the application.

Also, you must understand what the application is doing while it hangs. Is it waiting (low CPU usage), or it is looping forever (almost 100% CPU usage)? Most of the cases are a waiting problem.

You can, however, still identify two cases:

- Typical deadlock
- Standby condition while the application waits for a resource to arrive

An example of a standby condition is where the client sends a request to the server and stops while waiting for the reply. The default behavior of the ORB is to wait indefinitely.

You can set a couple of properties to avoid this condition:

- com.ibm.CORBA.LocateRequestTimeout
- com.ibm.CORBA.RequestTimeout

When the property com.ibm.CORBA.enableLocateRequest is set to true (the default is false), the ORB first sends a short message to the server to find the object that it needs to access. This first contact is the Locate Request. You must now set the LocateRequestTimeout to a value other than 0 (which is equivalent to infinity). A good value could be something around 5000 ms.

Also, set the RequestTimeout to a value other than 0. Because a reply to a request is often large, allow more time for the reply, such as 10,000 ms. These values are suggestions and might be too low for slow connections. When a request runs out of time, the client receives an explanatory CORBA exception.

When an application hangs, consider also another property that is called com.ibm.CORBA.FragmentTimeout. This property was introduced in IBM ORB 1.3.1, when the concept of fragmentation was implemented to increase performance. You can now split long messages into small chunks or fragments and send one after the other over the net. The ORB waits for 30 seconds (default value) for the next fragment before it throws an exception. If you set this property, you disable this timeout, and problems of waiting threads might occur.

If the problem seems to be a deadlock or hang, capture the Javadump information. After capturing the information, wait for a minute or so, and do it again. A comparison of the two snapshots shows whether any threads have changed state. For information about how to do this operation, see "Triggering a Javadump" on page 175.

In general, stop the application, enable the orb traces and restart the application. When the hang is reproduced, the partial traces that can be retrieved can be used by the IBM ORB service team to help understand where the problem is.

Running the client without the server running before the client is started

An example of the error messages that are generated from this process.

This operation outputs:

```
(org.omg.CORBA.COMM_FAILURE)
Hello Client exception:
        org.omg.CORBA.COMM_FAILURE:minor code:1 completed:No
             at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:145)
             at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:77)
             at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
             at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:75)
             at com.ibm.rmi.corba.ClientDelegate.createRequest(ClientDelegate.java:440)
             at com.ibm.rmi.corba.ClientDelegate.is_a(ClientDelegate.java:571)
             at org.omg.CORBA.portable.ObjectImpl._is_a(ObjectImpl.java:74)
             at org.omg.CosNaming.NamingContextHelper.narrow(NamingContextHelper.java:58)
             com.sun.jndi.cosnaming.CNCtx.callResolve(CNCtx.java:327)
```

Client and server are running, but not naming service

An example of the error messages that are generated from this process.

The output is:

```
Hello Client exception:Cannot connect to ORB
Javax.naming.CommunicationException:
        Cannot connect to ORB.Root exception is org.omg.CORBA.COMM_FAILURE  minor code:1 completed:No
               at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:145)
               at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:77)
               at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
               at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:75)
               at com.ibm.rmi.corba.ClientDelegate.createRequest(ClientDelegate.java:440)
               at com.ibm.rmi.corba.InitialNamingClient.resolve(InitialNamingClient.java:197)
               at com.ibm.rmi.corba.InitialNamingClient.cachedInitialReferences(InitialNamingClient.j
               at com.ibm.rmi.corba.InitialNamingClient.resolve_initial_references(InitialNamingClien
               at com.ibm.rmi.corba.ORB.resolve_initial_references(ORB.java:1269)
               .........
```

You must start the Java IDL name server before an application or applet starts that uses its naming service. Installation of the Java IDL product creates a script or executable file that starts the Java IDL name server.

Start the name server so that it runs in the background. If you do not specify otherwise, the name server listens on port 2809 for the bootstrap protocol that is used to implement the ORB resolve_initial_references() and list_initial_references() methods.

Specify a different port, for example, 1050, as follows:

```
tnameserv -ORBInitialPort 1050
```

Clients of the name server must be made aware of the new port number. Do this by setting the org.omg.CORBA.ORBInitialPort property to the new port number when you create the ORB object.

Running the client with MACHINE2 (client) unplugged from the network

An example of the error messages that are generated when the client has been unplugged form the network.

Your output is:

```
(org.omg.CORBA.TRANSIENT CONNECT_FAILURE)

Hello Client exception:Problem contacting address:corbaloc:iiop:machine2:2809/NameService
javax.naming.CommunicationException:Problem contacting address:corbaloc:iiop:machine2:2809/N
      is org.omg.CORBA.TRANSIENT:CONNECT_FAILURE (1)minor code:4942F301 completed:No
              at com.ibm.CORBA.transport.TransportConnectionBase.connect(TransportConnectionBase.jav
              at com.ibm.rmi.transport.TCPTransport.getConnection(TCPTransport.java:178)
              at com.ibm.rmi.iiop.TransportManager.get(TransportManager.java:79)
              at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:131)
              at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
              at com.ibm.CORBA.iiop.ClientDelegate._createRequest(ClientDelegate.java:2096)
              at com.ibm.CORBA.iiop.ClientDelegate.createRequest(ClientDelegate.java:1264)
              at com.ibm.CORBA.iiop.ClientDelegate.createRequest(ClientDelegate.java:1177)
              at com.ibm.rmi.corba.InitialNamingClient.resolve(InitialNamingClient.java:252)
              at com.ibm.rmi.corba.InitialNamingClient.cachedInitialReferences(InitialNamingClient.j
              at com.ibm.rmi.corba.InitialNamingClient.resolve_initial_references(InitialNamingClien
              at com.ibm.rmi.corba.InitialReferenceClient.resolve_initial_references(InitialReferenc
              at com.ibm.rmi.corba.ORB.resolve_initial_references(ORB.java:3211)
              at com.ibm.rmi.iiop.ORB.resolve_initial_references(ORB.java:523)
              at com.ibm.CORBA.iiop.ORB.resolve_initial_references(ORB.java:2898)
              ..........
```

IBM ORB service: collecting data

This section describes how to collect data about ORB problems.

If after all these verifications, the problem is still present, collect at all nodes of the problem the following:

- Operating system name and version.
- Output of `java -Xrealtime -version`
- Output of `java com.ibm.CORBA.iiop.Version`.
- Output of `rmic -iiop -version`, if rmic is involved.
- ASV build number (WebSphere Application Server only).
- If you think that the problem is a regression, include the version information for the most recent known working build and for the failing build.
- If this is a runtime problem, collect debug and communication traces of the failure from each node in the system (as explained earlier in this section).
- If the problem is in `rmic -iiop` or `rmic -idl`, set the options: `-J-Djavac.dump.stack=1 -Xtrace`, and capture the output.
- Typically this step is not necessary. If it looks like the problem is in the buffer fragmentation code, IBM service will return the defect asking for an additional set of traces, which you can produce by executing with `-Dcom.ibm.CORBA.FragmentSize=0`.

A testcase is not essential, initially. However, a working testcase that demonstrates the problem by using only the Java SDK classes will speed up the resolution time for the problem.

Preliminary tests

The ORB is affected by problems with the underlying network, hardware, and JVM.

When a problem occurs, the ORB can throw an org.omg.CORBA.* exception, some text that describes the reason, a minor code, and a completion status. Before you assume that the ORB is the cause of problem, ensure the following:

- The scenario can be reproduced in a similar configuration.
- The JIT is disabled (see "JIT and AOT problem determination" on page 241).
- No AOT compiled code is being used

Also:

- Disable additional CPUs.
- Disable Simultaneous Multithreading (SMT) where possible.
- Eliminate memory dependencies with the client or server. The lack of physical memory can be the cause of slow performance, apparent hangs, or crashes. To remove these problems, ensure that you have a reasonable headroom of memory.
- Check physical network problems (firewalls, comm links, routers, DNS name servers, and so on). These are the major causes of CORBA COMM_FAILURE exceptions. As a test, ping your own workstation name.
- If the application is using a database such as DB2, switch to the most reliable driver. For example, to isolate DB2 AppDriver, switch to Net Driver, which is slower and uses sockets, but is more reliable.

NLS problem determination

The JVM contains built-in support for different locales. This section provides an overview of locales, with the main focus on fonts and font management.

The topics are:

- "Overview of fonts"
- "Font utilities" on page 155
- "Common NLS problem and possible causes" on page 155

Overview of fonts

When you want to display text, either in SDK components (AWT or Swing), on the console or in any application, characters have to be mapped to glyphs.

A glyph is an artistic representation of the character, in some typographical style, and is stored in the form of outlines or bitmaps. Glyphs might not correspond one-for-one with characters. For instance, an entire character sequence can be represented as a single glyph. Also, a single character can be represented by more than one glyph (for example, in Indic scripts).

A font is a set of glyphs, where each glyph is encoded in a particular encoding format, so that the character to glyph mapping can be done using the encoded value. Almost all of the available Java fonts are encoded in Unicode and provide universal mappings for all applications.

The most commonly available font types are TrueType and OpenType fonts.

Font specification properties

Specify fonts according to the following characteristics:

Font family
Font family is a group of several individual fonts that are related in appearance. For example: Times, Arial, and Helvetica.

Font style
Font style specifies that the font be displayed in various faces. For example: Normal, Italic, and Oblique

Font variant
Font variant determines whether the font should be displayed in normal caps or in small caps. A particular font might contain only normal caps, only small caps, or both types of glyph.

Font weight
Font weight refers to the boldness or the lightness of the glyph to be used.

Font size
Font size is used to modify the size of the displayed text.

Fonts installed in the system

On Linux platforms
To see the fonts that are either installed in the system or available for an application to use, type the command: `xset -q ""`. If your **PATH** also points to the SDK (as it should be), `xset -q` output also shows the fonts that are bundled with the Developer Kit.

Use `xset +fp` to add the font path and `xset -fp` to remove the font path.

Font utilities

A list of font utilities that are supported.

Font utilities on Linux

xlsfonts
Use **xlsfonts** to check whether a particular font is installed on the system. For example: `xlsfonts | grep ksc` will list all the Korean fonts in the system.

iconv
Use to convert the character encoding from one encoding to other. Converted text is written to standard output. For example: `iconv -f oldset -t newset [file ...]`

Options are:

-f oldset
Specifies the source codeset (encoding).

-t newset
Specifies the destination codeset (encoding).

file
The file that contain the characters to be converted; if no file is specified, standard input is used.

Common NLS problem and possible causes

A common NLS problem with potential solutions.

Why do I see a square box or ??? (question marks) in the SDK components?
This effect is caused mainly because Java is not able to find the correct font file to display the character. If a Korean character should be displayed, the system should be using the Korean locale, so that Java can take the correct font file. If you are seeing boxes or queries, check the following:

For AWT components:

1. Check your locale with `locale`.
2. To change the locale, export `LANG=zh_TW` (for example)
3. If this still does not work, try to log in with the required language.

For Swing components:

1. Check your locale with `locale`
2. To change the locale, export `LANG=zh_TW` (for example)
3. If you know which font you have used in your application, such as serif, try to get the corresponding physical font by looking in the fontpath. If the font file is missing, try adding it there.

Attach API problem determination

This section helps you solve problems involving the Attach API.

The IBM Java Attach API uses shared semaphores, sockets, and file system artifacts to implement the attach protocol. Problems with these artifacts might adversely affect the operation of applications when they use the attach API.

Note: Error messages from agents on the target VM go to `stderr` or `stdout` for the target VM. They are not reported in the messages output by the attaching VM.

Deleting files in `/tmp`

The attach API depends on the contents of a common directory. By default the common directory is `/tmp/.com_ibm_tools_attach`. Problems are caused if you modify the common directory in one of the following ways:

- Deleting the common directory.
- Deleting the contents of the common directory.
- Changing the permissions of the common directory or any of its content.

If you do modify the common directory, possible effects include:

- Semaphore “leaks” might occur, where excessive numbers of unused shared semaphores are opened. You can remove the semaphores using the command:

  ```
  ipcrm -s <semid>
  ```

  Use the command to delete semaphores that have keys starting with “0xa1”.
- The Java VMs might not be able to list existing target VMs.
- The Java VMs might not be able to attach to existing target VMs.
- The Java VM might not be able to enable its attach API.

If the common directory cannot be used, a Java VM attempts to recreate the common directory. However, the JVM cannot recreate the files related to currently executing VMs.

The VirtualMachine.attach(String id) method reports `AttachNotSupportedException: No provider for virtual machine id`

There are several possible reasons for this message:

- The target VM might be owned by another userid. The attach API can only connect a VM to a target VM with the same userid.
- The attach API for the target VM might not have launched yet. There is a short delay from when the Java VM launches to when the attach API is functional.
- The attach API for the target VM might have failed. Verify that the directory `/tmp/.com_ibm_tools_attach/<id>` exists, and that the directory is readable and writable by the userid.
- The target directory `/tmp/.com_ibm_tools_attach/<id>` might have been deleted.
- The attach API might not have been able to open the shared semaphore. To verify that there is at least one shared semaphore, use the command:

  ```
  ipcs -s
  ```

  If there is a shared semaphore, at least one key starting with "0xa1" appears in the output from the ipcs command.

  Note: The number of available semaphores is limited on systems which use System V IPC, including Linux, z/OS®, and AIX®.

The VirtualMachine.attach() method reports `AttachNotSupportedException`

There are several possible reasons for this message:

- The target process is dead or suspended.
- The target process, or the hosting system is heavily loaded. The result is a delay in responding to the attach request.
- The network protocol has imposed a wait time on the port used to attach to the target. The wait time might occur after heavy use of the attach API, or other protocols which use sockets. To check if any ports are in the TIME_WAIT state, use the command:

  ```
  netstat -a
  ```

The VirtualMachine.loadAgent(), VirtualMachine.loadAgentLibrary(),, or VirtualMachine.loadAgentPath() methods report `com.sun.tools.attach.AgentLoadException` or `com.sun.tools.attach.AgentInitializationException`

There are several possible reasons for this message:

- The JVMTI agent or the agent JAR file might be corrupted. Try loading the agent at startup time using the -javaagent, -agentlib, or -agentpath option, depending on which method reported the problem.
- The agent might be attempting an operation which is not available after VM startup.

A process running as root can see a target using AttachProvider.listVirtualMachines(), but attempting to attach results in an `AttachNotSupportedException`

A process can attach only to processes owned by the same user. To attach to a non-root process from a root process, first use the su command to change the effective UID of the attaching process to the UID of the target UID, before attempting to attach.

Chapter 14. Using diagnostic tools

Diagnostics tools are available to help you solve your problems.

This section describes how to use the tools. The chapters are:

Note: JVMPI is now a deprecated interface, replaced by JVMTI.

Using dump agents

Dump agents are set up during JVM initialization. They enable you to use events occurring in the JVM, such as Garbage Collection, thread start, or JVM termination, to initiate dumps or to start an external tool.

The default dump agents are sufficient for most cases. Use the **-Xdump** option to add and remove dump agents for various JVM events, update default dump settings (such as the dump name), and limit the number of dumps that are produced.

This section describes:

Using the -Xdump option

The **-Xdump** option controls the way you use dump agents and dumps.

The **-Xdump** option allows you to:

- Add and remove dump agents for various JVM events.
- Update default dump agent settings.
- Limit the number of dumps produced.
- Show dump agent help.

You can have multiple **-Xdump** options on the command line and also multiple dump types triggered by multiple events. For example:

```
java -Xdump:heap:none -Xdump:heap+java:events=vmstart+vmstop <class> [args...]
```

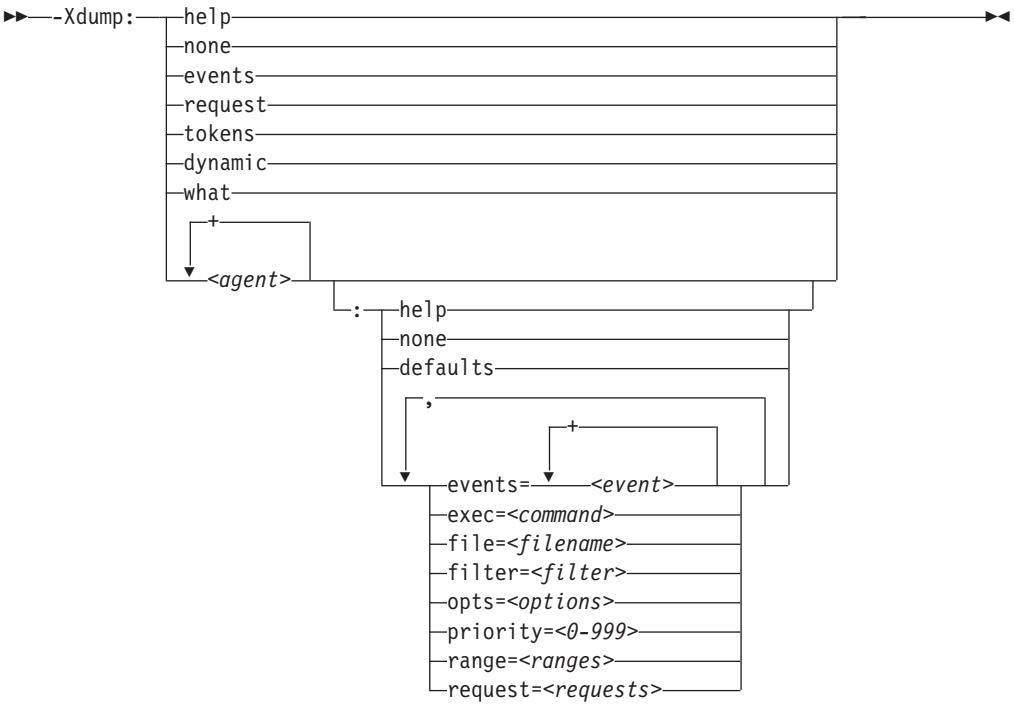
turns off all Heapdumps and create a dump agent that produces a Heapdump and a Javadump when either a vmstart or vmstop event occurs.

You can use the **-Xdump:what** option to list the registered dump agents. The registered dump agents listed might be different to those specified because the JVM ensures that multiple **-Xdump** options are merged into a minimum set of dump agents.

The events keyword is used as the prime trigger mechanism. However, you can use additional keywords to further control the dump produced.

The syntax of the **-Xdump** option is as follows:

-Xdump command-line option syntax



Users of UNIX style shells must be aware that unwanted shell expansion might occur because of the characters used in the dump agent options. To avoid unpredictable results, enclose this command line option in quotation marks. For example:

```
java "-Xdump:java:events=throw,filter=*Memory*" <Class>
```

For more information, see the manual for your shell.

Help options

These options display usage and configuration information for dumps, as shown in the following table:

Command	Result
-Xdump:help	Display general dump help
-Xdump:events	List available trigger events
-Xdump:request	List additional VM requests
-Xdump:tokens	List recognized label tokens
-Xdump:what	Show registered agents on startup
-Xdump:<*agent*>:help	Display detailed dump agent help
-Xdump:<*agent*>:defaults	Display default settings for this agent

Merging -Xdump agents

-Xdump agents are always merged internally by the JVM, as long as none of the agent settings conflict with each other.

If you configure more than one dump agent, each responds to events according to its configuration. However, the internal structures representing the dump agent configuration might not match the command line, because dump agents are merged for efficiency. Two sets of options can be merged as long as none of the agent settings conflict. This means that the list of installed dump agents and their parameters produced by **-Xdump:what** might not be grouped in the same way as the original **-Xdump** options that configured them.

For example, you can use the following command to specify that a dump agent collects a javadump on class unload:

```
java -Xdump:java:events=unload -Xdump:what
```

This command does not create a new agent, as can be seen in the results from the **-Xdump:what** option.

```
...
----------------------
-Xdump:java:
    events=gpf+user+abort+unload,
    label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
    range=1..0,
    priority=10,
    request=exclusive
----------------------
```

The configuration is merged with the existing javadump agent for events **gpf**, **user**, and **abort**, because none of the specified options for the new unload agent conflict with those for the existing agent.

In the above example, if one of the parameters for the unload agent is changed so that it conflicts with the existing agent, then it cannot be merged. For example, the following command specifies a different priority, forcing a separate agent to be created:

```
java -Xdump:java:events=unload,priority=100 -Xdump:what
```

The results of the **-Xdump:what** option in the command are as follows.

```
...
----------------------
-Xdump:java:
    events=unload,
    label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
    range=1..0,
```

```
    priority=100,
    request=exclusive
----------------------
-Xdump:java:
    events=gpf+user+abort,
    label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
    range=1..0,
    priority=10,
    request=exclusive
----------------------
```

To merge dump agents, the **request**, **filter**, **opts**, **label**, and **range** parameters must match exactly. If you specify multiple agents that filter on the same string, but keep all other parameters the same, the agents are merged. For example:

```
java -Xdump:none -Xdump:java:events=uncaught,filter=java/lang/NullPointerException \\
-Xdump:java:events=unload,filter=java/lang/NullPointerException -Xdump:what
```

The results of this command are as follows.

```
Registered dump agents
----------------------
-Xdump:java:
    events=unload+uncaught,
    filter=java/lang/NullPointerException,
    label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
    range=1..0,
    priority=10,
    request=exclusive
----------------------
```

Dump agents

A dump agent performs diagnostic tasks when triggered. Most dump agents save information on the state of the JVM for later analysis. The “tool” agent can be used to trigger interactive diagnostics.

The following table shows the dump agents:

Dump agent	Description
console	Basic thread dump to stderr.
system	Capture raw process image. See “Using system dumps and the dump viewer” on page 192.
tool	Run command-line program.
java	Write application summary. See “Using Javadump” on page 174.
heap	Capture heap graph. See “Using Heapdump” on page 187.
snap	Take a snap of the trace buffers.

Console dumps

Console dumps are very basic dumps, in which the status of every Java thread is written to stderr.

In this example, the **range=1..1** suboption is used to control the amount of output to just one thread start (in this case, the start of the Signal Dispatcher thread).

```
java -Xrealtime -Xdump:console:events=thrstart,range=1..1 -version

JVMDUMP006I Processing dump event "thrstart", detail "" - please wait.
```

```
-------- Console dump --------

Stack Traces of Threads:

ThreadName=Signal Dispatcher(08A23070)
Status=Running

ThreadName=main(08A22830)
Status=Waiting
Monitor=08AE6720 (Thread public flags mutex)
Count=0
Owner=(00000000)
In java/lang/reflect/Method.acquireMethodAccessor()V
In java/lang/reflect/Method.invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;
In com/ibm/misc/SystemIntialization.lastChanceHook()V
In java/lang/System.completeInitialization0()V
In java/lang/System.access$000()V
In java/lang/System$InitHelper.<clinit>()V
In java/lang/J9VMInternals.initializeImpl(Ljava/lang/Class;)V
In java/lang/J9VMInternals.initialize(Ljava/lang/Class;)V
In java/lang/System.completeInitialization()V
In java/lang/Thread.<init>(Ljava/lang/String;Ljava/lang/Object;IZ)V

^^^^^^^^ Console dump ^^^^^^^^
JVMDUMP013I Processed dump event "thrstart", detail "".</init></clinit>
```

Two threads are displayed in the dump because the main thread does not generate a thrstart event.

System dumps

System dumps involve dumping the address space and as such are generally very large.

The bigger the footprint of an application the bigger its dump. A dump of a major server-based application might take up many gigabytes of file space and take several minutes to complete.

```
java -Xrealtime -Xdump:system:events=vmstop -version

JVMDUMP006I Processing dump event "vmstop", detail "#00000000" - please wait.
JVMDUMP007I JVM Requesting System dump using '/home/user/core.20090612.150234.3247.0001.dmp'
JVMDUMP010I System dump written to /home/user/core.20090612.150234.3247.0001.dmp
JVMDUMP013I Processed dump event "vmstop", detail "#00000000".
```

See “Using system dumps and the dump viewer” on page 192 for more information about analyzing a system dump.

Related information

“Using system dumps and the dump viewer” on page 192
The JVM can generate native system dumps, also known as core dumps, under configurable conditions. System dumps are typically quite large. Use the gdb tool to analyze a system dump on Linux.

Tool option

The **tool** option allows external processes to be started when an event occurs.

The following example displays a simple message when the JVM stops. The `%pid` token is used to pass the pid of the process to the command. The list of available tokens can be printed with **-Xdump:tokens**, or found in “Dump agent tokens” on page 170. If you do not specify a tool to use, a platform specific debugger is started.

```
java -Xrealtime -Xdump:tool:events=vmstop,exec="echo process %pid has finished" -version

 VMDUMP006I Processing dump event "vmstop", detail "#00000000" - please wait.
 JVMDUMP007I JVM Requesting Tool dump using 'echo process 254050 has finished'
 JVMDUMP011I Tool dump spawned process 344292
 process 254050 has finished
 JVMDUMP013I Processed dump event "vmstop", detail "#00000000".
```

By default, the **range** option is set to 1..1. If you do not specify a range option for the dump agent the tool will be started once only. To start the tool every time the event occurs, set the **range** option to 1..0. See “range option” on page 169 for more information.

Javadumps

Javadumps are an internally generated and formatted analysis of the JVM, giving information that includes the Java threads present, the classes loaded, and heap statistics.

An example of producing a Javadump when a class is loaded is shown below.

```
java -Xrealtime -Xdump:java:events=load,filter=java/lang/String -version

 JVMDUMP006I Processing dump event "load", detail "java/lang/String" - please wait.
 JVMDUMP007I JVM Requesting Java dump using '/home/user/javacore.20090602.094449.274632.0001.txt'
 JVMDUMP010I Java dump written to /home/user/javacore.20090602.094449.274632.0001.txt
 JVMDUMP013I Processed dump event "load", detail "java/lang/String".
```

See “Using Javadump” on page 174 for more information about analyzing a Javadump.

Related information

“Using Javadump” on page 174
Javadump produces files that contain diagnostic information related to the JVM and a Java application captured at a point during execution. For example, the information can be about the operating system, the application environment, threads, stacks, locks, and memory.

Heapdumps

Heapdumps produce phd format files by default.

“Using Heapdump” on page 187 provides more information about Heapdumps. The following example shows the production of a Heapdump. In this case, both a phd and a classic (.txt) Heapdump have been requested by the use of the **opts=** option.

```
java -Xrealtime -Xdump:heap:events=vmstop,opts=PHD+CLASSIC -version

JVMDUMP006I Processing dump event "vmstop", detail "#00000000" - please wait.
JVMDUMP007I JVM Requesting Heap dump using '/home/user/heapdump.20090602.095239.164050.0001.phd'
JVMDUMP010I Heap dump written to /home/user/heapdump.20090602.095239.164050.0001.phd
JVMDUMP007I JVM Requesting Heap dump using '/home/user/heapdump.20090602.095239.164050.0001.txt'
JVMDUMP010I Heap dump written to /home/user/heapdump.20090602.095239.164050.0001.txt
JVMDUMP013I Processed dump event "vmstop", detail "#00000000".
```

See “Using Heapdump” on page 187 for more information about analyzing a Heapdump.

Related information

"Using Heapdump" on page 187
The term Heapdump describes the IBM Virtual Machine for Java mechanism that generates a dump of all the live objects that are on the Java heap; that is, those that are being used by the running Java application.

Snap traces

Snap traces are controlled by **-Xdump**. They contain the tracepoint data held in the trace buffers.

The example below shows the production of a snap trace.

```
java -Xrealtime -Xdump:snap:events=vmstop -version

 JVMDUMP006I Processing dump event "vmstop", detail "#00000000" - please wait.
 JVMDUMP007I JVM Requesting Snap dump using '/home/user/Snap.20090603.063646.315586.0001.trc'
 JVMDUMP010I Snap dump written to /home/user/Snap.20090603.063646.315586.0001.trc
 JVMDUMP013I Processed dump event "vmstop", detail "#00000000".
```

Snap traces require the use of the trace formatter for further analysis.

See "Using the trace formatter" on page 229 for more information about analyzing a snap trace.

Dump events

Dump agents are triggered by events occurring during JVM operation.

Some events can be filtered to improve the relevance of the output. See "filter option" on page 167 for more information.

Note: The unload and expand events currently do not occur in WebSphere Real Time. Classes are in immortal memory and cannot be unloaded.

Note: The gpf and abort events cannot trigger a heap dump, prepare the heap (request=prepwalk), or compact the heap (request=compact).

The table below shows events available as dump agent triggers:

Event	Triggered when...	Filter operation
gpf	A General Protection Fault (GPF) occurs.	
user	The JVM receives the SIGQUIT (Linux) signal from the operating system.	
abort	The JVM receives the SIGABRT signal from the operating system.	
vmstart	The virtual machine is started.	
vmstop	The virtual machine stops.	Filters on exit code; for example, **filter=#129..#192#-42#255**
load	A class is loaded.	Filters on class name; for example, **filter=java/lang/String**
unload	A class is unloaded.	
throw	An exception is thrown.	Filters on exception class name; for example, **filter=java/lang/OutOfMem***
catch	An exception is caught.	Filters on exception class name; for example, **filter=*Memory***

Event	Triggered when...	Filter operation
uncaught	A Java exception is not caught by the application.	Filters on exception class name; for example, **filter=*MemoryError**
systhrow	A Java exception is about to be thrown by the JVM. This is different from the 'throw' event because it is only triggered for error conditions detected internally in the JVM.	Filters on exception class name; for example, **filter=java/lang/OutOfMem***
thrstart	A new thread is started.	
blocked	A thread becomes blocked.	
thrstop	A thread stops.	
fullgc	A garbage collection cycle is started.	
slow	A thread takes longer than 5ms to respond to an internal JVM request.	Changes the time taken for an event to be considered slow; for example, **filter=#300ms** will trigger when a thread takes longer than 300ms to respond to an internal JVM request.

Advanced control of dump agents

Options are available to give you more control over dump agent behavior.

exec option

The exec option is used by the tool dump agent to specify an external application to start.

See “Tool option” on page 163 for an example and usage information.

file option

The file option is used by dump agents that write to a file.

It specifies where the diagnostics information should be written. For example:

```
java -Xrealtime -Xdump:heap:events=vmstop,file=my.dmp
```

You can use tokens to add context to dump file names. See “Dump agent tokens” on page 170 for more information.

The location for the dump is selected from these options, in this order:

1. The location specified on the command line.
2. The location specified by the relevant environment variable.
   - **IBM_JAVACOREDIR** for Javadump.
   - **IBM_HEAPDUMPDIR** for Heapdump.
   - **IBM_COREDIR** for system dump, .
   - **IBM_COREDIR** for snap traces, .
3. The current working directory of the JVM process.

If the directory does not exist, it will be created.

If the dump cannot be written to the selected location, the JVM will fall-back to the following locations, in this order:

1. The location specified by the **TMPDIR** environment variable.
2. The `/tmp` directory.

filter option

Some JVM events occur thousands of times during the lifetime of an application. Dump agents can use filters and ranges to avoid excessive dumps being produced.

Wildcards

You can use a wildcard in your exception event filter by placing an asterisk only at the beginning or end of the filter. The following command does not work because the second asterisk is not at the end:

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#*.myVirtualMethod
```

In order to make this filter work, it must be changed to:

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#MyApplication.*
```

Class loading and exception events

You can filter class loading (load) and exception (throw, catch, uncaught, systhrow) events by Java class name:

```
-Xdump:java:events=throw,filter=java/lang/OutOfMem*
-Xdump:java:events=throw,filter=*MemoryError
-Xdump:java:events=throw,filter=*Memory*
```

From Java 6 SR 3, you can filter throw, uncaught, and systhrow exception events by Java method name:

```
-Xdump:java:events=throw,filter=ExceptionClassName[#ThrowingClassName.throwingMethodName[#stackFra
```

Optional portions are shown in square brackets.

From Java 6 SR 3, you can filter the catch exception events by Java method name:

```
-Xdump:java:events=catch,filter=ExceptionClassName[#CatchingClassName.catchingMethodName]
```

Optional portions are shown in square brackets.

vmstop event

You can filter the JVM shut down event by using one or more exit codes:

```
-Xdump:java:events=vmstop,filter=#129..192#-42#255
```

slow event

You can filter the slow event to change the time threshold from the default of 5 ms:

```
-Xdump:java:events=slow,filter=#300ms
```

You cannot set the filter to a time lower than the default time.

allocation event

You must filter the allocation event to specify the size of objects that cause a trigger. You can set the filter size from zero up to the maximum value of a 32 bit pointer on 32 bit platforms, or the maximum value of a 64 bit pointer on 64 bit platforms. Setting the lower filter value to zero triggers a dump on all allocations.

For example, to trigger dumps on allocations greater than 5 Mb in size, use:

```
-Xdump:stack:events=allocation,filter=#5m
```

To trigger dumps on allocations between 256Kb and 512Kb in size, use:

```
-Xdump:stack:events=allocation,filter=#256k..512k
```

The allocation event is available from Java 6 SR 5 onwards.

Other events

If you apply a filter to an event that does not support filtering, the filter is ignored.

opts option

The Heapdump agent uses this option to specify the type of file to produce.

Heapdumps and the opts option

You can specify a PHD Heapdump, a classic text Heapdump, or both. For example:

–Xdump:heap:opts=PHD (default)
–Xdump:heap:opts=CLASSIC
–Xdump:heap:opts=PHD+CLASSIC

See “Enabling text formatted ("classic") Heapdumps” on page 188 for more information.

The ceedump agent is the preferred way to specify LE dumps, for example:

```
-Xdump:ceedump:events=gpf
```

Priority option

One event can generate multiple dumps. The agents that produce each dump run sequentially and their order is determined by the priority keyword set for each agent.

Examination of the output from **-Xdump:what** shows that a gpf event produces a snap trace, a Javadump, and a system dump. In this example, the system dump will run first (priority 999), the snap dump second (priority 500), and the Javadump last (priority 10):

–Xdump:heap:events=vmstop,priority=123

The maximum value allowed for priority is 999. Higher priority dump agents will be started first.

If you do not specifically set a priority, default values are taken based on the dump type. The default priority and the other default values for a particular type of dump, can be displayed by using **-Xdump:<*type*>:defaults**. For example:

```
java -Xrealtime -Xdump:heap:defaults -version

Default -Xdump:heap settings:

  events=gpf+user
  filter=
  file=/home/user/heapdump.%Y%m%d.%H%M%S.%pid.phd
  range=1..0
  priority=40
  request=exclusive+prepwalk
  opts=PHD
```

range option

You can start and stop dump agents on a particular occurrence of a JVM event by using the range suboption.

For example:

```
-Xdump:java:events=fullgc,range=100..200
```

Note: range=1..0 against an event means "on every occurrence".

The JVM default dump agents have the **range** option set to 1..0 for all events except `systhrow`. All `systhrow` events with `filter=java/lang/OutOfMemoryError` have the **range** set to 1..4, which limits the number of dumps produced on `OutOfMemory` conditions to a maximum of 4. For more information, see "Default dump agents" on page 170

If you add a new dump agent and do not specify the range, a default of 1..0 is used.

request option

Use the request option to ask the JVM to prepare the state before starting the dump agent.

The available options are listed in the following table:

Option value	Description
exclusive	Request exclusive access to the JVM.
compact	Run garbage collection. This option removes all unreachable objects from the heap before the dump is generated.
prepwalk	Prepare the heap for walking. You must also specify **exclusive** when using this option.
serial	Suspend other dumps until this one has completed.
multiple	Produce separate heap dumps for each RTSJ memory area.

In general, the default request options are sufficient.

You can specify more than one request option using +. For example:

-Xdump:heap:request=exclusive+compact+prepwalk+multiple

defaults option

Each dump type has default options. To view the default options for a particular dump type, use **-Xdump:<type>:defaults**.

You can change the default options at runtime. For example, you can direct Java dump files into a separate directory for each process, and guarantee unique files by adding a sequence number to the file name using:

-Xdump:java:defaults:file=dumps/%pid/javacore-%seq.txt

This option does not add a Javadump agent; it updates the default settings for Javadump agents. Further Javadump agents will then create dump files using this specification for filenames, unless overridden.

Note: Changing the defaults for a dump type will also affect the default agents for that dump type added by the JVM during initialization. For example if you change the default file name for Javadumps, that will change the file name used by the default Javadump agents. However, changing the default **range** option will not change the range used by the default Javadump agents, because those agents override the **range** option with specific values.

Dump agent tokens

Use tokens to add context to dump file names and to pass command-line arguments to the tool agent.

The tokens available are listed in the following table:

Token	Description
%Y	Year (4 digits)
%y	Year (2 digits)
%m	Month (2 digits)
%d	Day of the month (2 digits)
%H	Hour (2 digits)
%M	Minute (2 digits)
%S	Second (2 digits)
%pid	Process id
%uid	User name
%seq	Dump counter
%tick	msec counter
%home	Java home directory
%last	Last dump

Default dump agents

The JVM adds a set of dump agents by default during its initialization. You can override this set of dump agents using **-Xdump** on the command line.

See “Removing dump agents” on page 171. for more information.

Use the **-Xdump:what** option on the command line to show the registered dump agents. The sample output shows the default dump agents that are in place:

```
java -Xrealtime -Xdump:what

Registered dump agents
----------------------
-Xdump:system:
    events=gpf+abort,
    label=/home/user/core.%Y%m%d.%H%M%S.%pid.%seq.dmp,
    range=1..0,
    priority=999,
    request=serial
----------------------
-Xdump:snap:
    events=gpf+abort,
    label=/home/user/Snap%seq.%Y%m%d.%H%M%S.%pid.%seq.trc,
    range=1..0,
    priority=500,
```

```
    request=serial
----------------------
-Xdump:snap:
    events=systhrow,
    filter=java/lang/OutOfMemoryError,
    label=/home/user/Snap%seq.%Y%m%d.%H%M%S.%pid.%seq.trc,
    range=1..4,
    priority=500,
    request=serial
----------------------
-Xdump:heap:
    events=systhrow,
    filter=java/lang/OutOfMemoryError,
    label=/home/user/heapdump.%Y%m%d.%H%M%S.%pid.%seq.phd,
    range=1..4,
    priority=40,
    request=exclusive+prepwalk+compact,
    opts=PHD
----------------------
-Xdump:java:
    events=gpf+user+abort,
    label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
    range=1..0,
    priority=10,
    request=exclusive
----------------------
-Xdump:java:
    events=systhrow,
    filter=java/lang/OutOfMemoryError,
    label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
    range=1..4,
    priority=10,
    request=exclusive
----------------------
```

Removing dump agents

You can remove all default dump agents and any preceding dump options by using **-Xdump:none**.

Use this option so that you can subsequently specify a completely new dump configuration.

You can also remove dump agents of a particular type. For example, to turn off all Heapdumps (including default agents) but leave Javadump enabled, use the following option:

-Xdump:java+heap:events=vmstop -Xdump:heap:none

If you remove all dump agents using **-Xdump:none** with no further **-Xdump** options, the JVM still provides these basic diagnostics:

- If a user signal (`kill -QUIT`) is sent to the JVM, a brief listing of the Java threads including their stacks, status, and monitor information is written to stderr.
- If a crash occurs, information about the location of the crash, JVM options, and native and Java stack traces are written to stderr. A system dump is also written to the user's home directory.

Tip: Removing dump agents and specifying a new dump configuration can require a long set of command-line options. To reuse command-line options, save the new dump configuration in a file and use the **-Xoptionsfile** option.

Dump agent environment variables

The **-Xdump** option on the command line is the preferred method for producing dumps for cases where the default settings are not enough. You can also produce dumps using the **JAVA_DUMP_OPTS** environment variable.

If you set agents for a condition using the **JAVA_DUMP_OPTS** environment variable, default dump agents for that condition are disabled; however, any **-Xdump** options specified on the command line will be used.

The **JAVA_DUMP_OPTS** environment variable is used as follows:

```
JAVA_DUMP_OPTS="ON<condition>(<agent>[<count>],<agent>[<count>]),ON<condition>(<agent>[<count>],...)
```

where:

- <*condition*> can be:
  - ANYSIGNAL
  - DUMP
  - ERROR
  - INTERRUPT
  - EXCEPTION
  - OUTOFMEMORY
- <*agent*> can be:
  - ALL
  - NONE
  - JAVADUMP
  - SYSDUMP
  - HEAPDUMP
- <*count*> is the number of times to run the specified agent for the specified condition. This value is optional. By default, the agent will run every time the condition occurs. This option is introduced in Java 6 SR2.

JAVA_DUMP_OPTS is parsed by taking the leftmost occurrence of each condition, so duplicates are ignored. The following setting will produce a system dump for the first error condition only:

```
ONERROR(SYSDUMP[1]),ONERROR(JAVADUMP)
```

Also, the `ONANYSIGNAL` condition is parsed before all others, so

```
ONINTERRUPT(NONE),ONANYSIGNAL(SYSDUMP)
```

has the same effect as

```
ONANYSIGNAL(SYSDUMP),ONINTERRUPT(NONE)
```

If the **JAVA_DUMP_TOOL** environment variable is set, that variable is assumed to specify a valid executable name and is parsed for replaceable fields, such as %pid. If %pid is detected in the string, the string is replaced with the JVM's own process ID. The tool specified by **JAVA_DUMP_TOOL** is run after any system dump or Heapdump has been taken, before anything else.

From Java 6 SR 2, the dump settings are applied in the following order, with the settings later in the list taking precedence:

1. Default JVM dump behavior.

2. **-Xdump** command-line options that specify **-Xdump:<type>:defaults**, see "defaults option" on page 169.
3. **DISABLE_JAVADUMP**, **IBM_HEAPDUMP**, and **IBM_HEAP_DUMP** environment variables.
4. **IBM_JAVADUMP_OUTOFMEMORY** and **IBM_HEAPDUMP_OUTOFMEMORY** environment variables.
5. **JAVA_DUMP_OPTS** environment variable.
6. Remaining **-Xdump** command-line options.

Prior to Java 6 SR 2, the **DISABLE_JAVADUMP**, **IBM_HEAPDUMP**, and **IBM_HEAP_DUMP** environment variables took precedence over the **JAVA_DUMP_OPTS** environment variable.

From Java 6 SR 2, setting **JAVA_DUMP_OPTS** only affects those conditions you specify. Actions on other conditions are left unchanged. Prior to Java 6 SR 2, setting **JAVA_DUMP_OPTS** overrides settings for all the conditions.

Signal mappings

The signals used in the **JAVA_DUMP_OPTS** environment variable map to multiple operating system signals.

The mapping of operating system signals to the "condition" when you are setting the **JAVA_DUMP_OPTS** environment variable is as follows:

EXCEPTION	SIGTRAP
	SIGILL
	SISEGV
	SIGFPE
	SIGBUS
	SIGXCPU
	SIGXFSZ
INTERRUPT	SIGINT
	SIGTERM
	SIGHUP
ERROR	SIGABRT
DUMP	SIGQUIT

Dump agent default locations

Dump output is written to different files, depending on the type of the dump. File names include a time stamp.

- **System dumps**: Output is written to a file named `core.%Y%m%d.%H%M%S.%pid.dmp`.
- **Javadumps**: Output is written to a file named `javacore.%Y%m%d.%H%M%S.%pid.%seq.txt`. See "Using Javadump" on page 174 for more information.
- **Heapdumps**: Output is written to a file named `heapdump.%Y%m%d.%H%M%S.%pid.phd`. See "Using Heapdump" on page 187 for more information.

Disabling dump agents with -Xrs

When using a debugger such as GDB or WinDbg to diagnose problems in JNI code, you might want to disable the signal handler of the Java runtime so that any signals received are handled by the operating system.

Using the **-Xrs** command-line option prevents the Java runtime handling exception signals such SIGSEGV and SIGABRT. When the Java runtime signal handler is disabled, a SIGSEGV or GPF crash does not call the JVM dump agents. Instead, dumps are produced depending on the operating system.

Disabling dump agents in Linux

If configured correctly, most Linux distributions produce a core file called `core.pid` in the process working directory when a process crashes. See "Setting up and checking your Linux environment" on page 125 for details on the required system configuration. Core dumps produced natively by Linux can be processed with jextract and analyzed with tools such as jdmpview and DTFJ. The Linux operating system core dump might not contain all the information included in a core dump produced by the JVM dump agents.

Using Javadump

Javadump produces files that contain diagnostic information related to the JVM and a Java application captured at a point during execution. For example, the information can be about the operating system, the application environment, threads, stacks, locks, and memory.

By default, a Javadump occurs when the JVM terminates unexpectedly. A Javadump can also be triggered by sending specific signals to the JVM. Javadumps are human readable.

The preferred way to control the production of Javadumps is by enabling dump agents (see "Using dump agents" on page 159) using **–Xdump:java:** on application startup. You can also control Javadumps by the use of environment variables. See "Environment variables and Javadump" on page 186.

Default agents are in place that (if not overridden) create Javadumps when the JVM terminates unexpectedly or when an out-of-memory exception occurs. Javadumps are also triggered by default when specific signals are received by the JVM.

Note: Javadump is also known as **Javacore**. Javacore is NOT the same as a **core file**, which is generated by a system dump.

This chapter describes:

Related information

"Javadumps" on page 164
Javadumps are an internally generated and formatted analysis of the JVM, giving information that includes the Java threads present, the classes loaded, and heap statistics.

Enabling a Javadump

Javadumps are enabled by default. You can turn off the production of Javadumps with **-Xdump:java:none**.

You are not recommended to turn off Javadumps because they are an essential diagnostics tool.

Use the **-Xdump:java** option to give more fine-grained control over the production of Javadumps. See "Using dump agents" on page 159 for more information.

Triggering a Javadump

Javadumps are triggered by a number of events, both in error situations and user-initiated.

By default, a Javadump is triggered when one of the following error conditions occurs:

A fatal native exception
Not a Java Exception. A "fatal" exception is one that causes the JVM to stop. The JVM handles this by producing a system dump followed by a snap trace file, a Javadump, and then terminating the process.

The JVM has insufficient memory to continue operation
There are many reasons for running out of memory. See Chapter 13, "Problem determination," on page 123 for more information.

You can also initiate a Javadump to obtain diagnostic information in one of the following ways:

You can send a signal to the JVM from the command line

The signal for Linux is `SIGQUIT`. Use the command `kill -QUIT n` to send the signal to a process with process id (PID) `n`. Alternatively, press **CTRL+** in the shell window that started Java. (**CTRL+V** on z/OS.)

The JVM will continue operation after the signal has been handled.

You can use the JavaDump() method in your application

The com.ibm.jvm.Dump class contains a static JavaDump() method that causes Java code to initiate a Javadump. In your application code, add a call to com.ibm.jvm.Dump.JavaDump(). This call is subject to the same Javadump environment variables that are described in "Enabling a Javadump."

The JVM will continue operation after the JavaDump has been produced.

You can initiate a Javadump using the wasadmin utility

In a WebSphere Application Server environment, use the wasadmin utility to initiate a dump.

The JVM will continue operation after the JavaDump has been produced.

You can configure a dump agent to trigger a Javadump

Use the **-Xdump:java:** option to configure a dump agent on the command line. See "Using the -Xdump option" on page 159 for more information.

You can use the trigger trace option to generate a Javadump

Use the **-Xtrace:trigger** option to produce a Javadump when the substring method shown in the following example is called:

```
-Xtrace:trigger=method{java/lang/String.substring,javadump}
```

For a detailed description of this trace option, see "trigger=<*clause*>[,<*clause*>][,<*clause*>]..." on page 226

Interpreting a Javadump

This section gives examples of the information contained in a Javadump and how it can be useful in problem solving.

The content and range of information in a Javadump might change between JVM versions or service refreshes. Some information might be missing, depending on the operating system platform and the nature of the event that produced the Javadump.

Javadump tags

The Javadump file contains sections separated by eyecatcher title areas to aid readability of the Javadump.

The first such eyecatcher is shown as follows:

```
NULL           ------------------------------------------------------------------------
0SECTION       ENVINFO subcomponent dump routine
NULL           ================================
```

Different sections contain different tags, which make the file easier to parse for performing simple analysis.

You can also use DTFJ to parse a Javadump, see "Using the Diagnostic Tool Framework for Java" on page 282 for more information.

An example tag (`1CIJAVAVERSION`) is shown as follows:

```
1CIJAVAVERSION J2RE 6.0 IBM J9 2.5 AIX ppc-32 build jvmap32srt60sr2-20090513_35395
```

Normal tags have these characteristics:

- Tags are up to 15 characters long (padded with spaces).
- The first digit is a nesting level (0,1,2,3).
- The second and third characters identify the section of the dump. The major sections are:
  - **CI** Command-line interpreter
  - **CL** Class loader
  - **LK** Locking
  - **ST** Storage (Memory management)
  - **TI** Title
  - **XE** Execution engine
- The remainder is a unique string, `JAVAVERSION` in the previous example.

Special tags have these characteristics:

- A tag of `NULL` means the line is just to aid readability.

- Every section is headed by a tag of `0SECTION` with the section title.

Here is an example of some tags taken from the start of a dump. The components are highlighted for clarification.

```
NULL           ------------------------------------------------------------------------
0SECTION       TITLE subcomponent dump routine
NULL           ===============================
1TISIGINFO     Dump Event "user" (00004000) received
1TIDATETIME    Date: 2009/06/03 at 06:54:19
1TIFILENAME    Javacore filename: /home/user/javacore.20090603.065419.315480.0001.txt
NULL           ------------------------------------------------------------------------
0SECTION       GPINFO subcomponent dump routine
NULL           ===============================
2XHOSLEVEL     OS Level : AIX 6.1
2XHCPUS        Processors -
3XHCPUARCH       Architecture : ppc
3XHNUMCPUS       How Many : 8
3XHNUMASUP       NUMA is either not supported or has been disabled by user
```

For the rest of the topics in this section, the tags are removed to aid readability.

TITLE, GPINFO, and ENVINFO sections

At the start of a Javadump, the first three sections are the TITLE, GPINFO, and ENVINFO sections. They provide useful information about the cause of the dump.

The following example shows some output taken from a simple Java test program using the -Xtrace option, that deliberately causes a “general protection fault” (GPF).

`TITLE`

Shows basic information about the event that caused the generation of the Javadump, the time it was taken, and its name.

`GPINFO`

Varies in content depending on whether the Javadump was produced because of a GPF or not. It shows some general information about the operating system. If the failure was caused by a GPF, GPF information about the failure is provided, in this case showing that the protection fault occurred in module `./myNative`. The registers specific to the processor and architecture are also displayed.

The `GPINFO` section also refers to the vmState, recorded in the console output as `VM flags`. The vmState is the thread-specific state of what was happening in the JVM at the time of the crash. The value for vmState is a 32-bit hexadecimal number of the format MMMMSSSS, where MMMM is the major component and SSSS is component specific code.

Major component	Code number
INTERPRETER	0x10000
GC	0x20000
GROW_STACK	0x30000
JNI	0x40000
JIT_CODEGEN	0x50000
BCVERIFY	0x60000
RTVERIFY	0x70000
SHAREDCLASSES	0x80000

In the following example, the value for vmState is `VM flags:00040000`, which indicates a crash in the JNI component.

When the vmState major component is JNI, the crash might be caused by customer JNI code or by Java SDK JNI code. Check the Javadump to reveal which JNI routine was called at the point of failure. The JNI is the only component where a crash might be caused by customer code.

When the vmState major component is JIT_CODEGEN, see the information at "JIT and AOT problem determination" on page 241.

ENVINFO

Shows information about the JRE level that failed and details about the command line that launched the JVM process and the JVM environment in place.

```
0SECTION TITLE subcomponent dump routine
NULL ===============================
1TISIGINFO Dump Event "gpf" (00002000) received
1TIDATETIME Date: 2009/06/09 at 09:10:19
1TIFILENAME Javacore filename: /home/test/javacore.20090609.091012.27334.0003.txt
NULL ------------------------------------------------------------------------
0SECTION GPINFO subcomponent dump routine
NULL ===============================
2XHOSLEVEL OS Level : Linux 2.6.16-rtj12.12smp
2XHCPUS Processors -
3XHCPUARCH Architecture : x86
3XHNUMCPUS How Many : 4
3XHNUMASUP NUMA is either not supported or has been disabled by user
NULL
1XHEXCPCODE J9Generic_Signal_Number: 00000004
1XHEXCPCODE Signal_Number: 0000000B
1XHEXCPCODE Error_Value: 00000000
1XHEXCPCODE Signal_Code: 00000001
1XHEXCPCODE Handler1: B7728EBA
1XHEXCPCODE Handler2: B77002A5
1XHEXCPCODE InaccessibleAddress: 00000000
NULL
1XHEXCPMODULE Module: ./myNative
1XHEXCPMODULE Module_base_address: A59EE000
1XHEXCPMODULE Symbol: Java_myNativeCrash_Crash
1XHEXCPMODULE Symbol_address: A59EE54C
NULL
1XHREGISTERS Registers:
2XHREGISTER EDI: A59EE54C
2XHREGISTER ESI: 00000000
2XHREGISTER EAX: 00000000
2XHREGISTER EBX: 00000088
2XHREGISTER ECX: 000000AC
2XHREGISTER EDX: 00000000
2XHREGISTER EIP: A59EE55C
2XHREGISTER ES: 0000007B
2XHREGISTER DS: 0000007B
2XHREGISTER ESP: B7FB003C
2XHREGISTER EFlags: 00010296
2XHREGISTER CS: 00000073
2XHREGISTER SS: 0000007B
2XHREGISTER EBP: B7FB0044
NULL
1XHFLAGS VM flags:00040000
NULL
NULL ------------------------------------------------------------------------
0SECTION ENVINFO subcomponent dump routine
NULL ================================
1CIJAVAVERSION J2RE 6.0 IBM J9 2.5 Linux x86-32 build jvmxi32rt60sr2-20090605_36710
1CIVMVERSION VM build 20090605_036710
1CIJITVERSION JIT enabled, AOT enabled - r10_20090603_1712
1CIGCVERSION GC - 20090601_AA
1CIRUNNINGAS Running as a standalone JVM
1CICMDLINE sdk/jre/bin/java -Xrealtime myNativeCrash
1CIJAVAHOMEDIR Java Home Dir: /home/test/sdk/jre
1CIJAVADLLDIR Java DLL Dir: /home/test/sdk/jre/bin
1CISYSCP Sys Classpath: /home/test/sdk/jre/lib/i386/realtime/jclSC160/realtime.jar....
1CIUSERARGS UserArgs:
2CIUSERARG -Xjcl:jclscar_25
```

```
2CIUSERARG -Dcom.ibm.oti.vm.bootstrap.library.path=/home/test/sdk/jre/lib/i386/realtime:/home/test/sdk/jre/lib/i386
2CIUSERARG -Dsun.boot.library.path=/home/test/sdk/jre/lib/i386/realtime:/home/test/sdk/jre/lib/i386
2CIUSERARG -Djava.library.path=/home/test/sdk/jre/lib/i386/realtime:/home/test/sdk/jre/lib/i386:.:/usr/lib
2CIUSERARG -Djava.home=/home/test/sdk/jre
2CIUSERARG -Djava.ext.dirs=/home/test/sdk/jre/lib/ext
2CIUSERARG -Duser.dir=/home/test
2CIUSERARG _j2se_j9=1119744 0xB77AC1E0
2CIUSERARG -Xdump
2CIUSERARG -Djava.class.path=.
2CIUSERARG -Xrealtime
2CIUSERARG -Dsun.java.command=myNativeCrash
2CIUSERARG -Dsun.java.launcher=SUN_STANDARD
2CIUSERARG -Dsun.java.launcher.pid=27334
2CIUSERARG _port_library 0xB77AE600
2CIUSERARG _org.apache.harmony.vmi.portlib 0x0805C998
```

In the example above, the following lines show where the crash occurred:

```
1XHEXCPMODULE Module: ./myNative
1XHEXCPMODULE Module_base_address: A59EE000
1XHEXCPMODULE Symbol: Java_myNativeCrash_Crash
1XHEXCPMODULE Symbol_address: A59EE54C
```

Storage Management (MEMINFO)

The MEMINFO section provides information about the Memory Manager.

See Using the Metronome Garbage Collector for details about how the memory manager component works.

This part of the Javadump gives various storage management values (in hexadecimal), including the free space and current size of the heap, immortal, and scoped memory areas. It also contains garbage collection history data, described in “Default memory management tracing” on page 210. Garbage collection history data is shown as a sequence of tracepoints, each with a timestamp, ordered with the most recent tracepoint first.

Javadumps produced by the standard JVM contain a “GC History” section. This information is not contained in Javadumps produced when using the Real Time JVM. Use the **-verbose:gc** option or the JVM snap trace to obtain information about GC behavior. See “Using verbose:gc information” on page 51 and “Snap traces” on page 165 for more details.

If you are running a program which uses scoped memory, you might find that if an OutOfMemoryError is thrown, none of the memory areas listed in the Javadump seem to be empty. This is because when a scope which is nested inside another scope runs out of memory, by the time the Javadump is generated, the inner scope could have been deleted. To get information which relates to the state of the memory areas at the time the OutOfMemoryError is thrown, run your program with the following command-line option:

```
-Xdump:java:events=throw,filter=java/lang/OutOfMemoryError,range=1..1
```

This generates a Javadump (additional to any that are later generated by default dump agents) when the OutOfMemoryError is thrown, rather than when the uncaught exception is detected (which happens slightly later). In this Javadump you should be able to see all memory areas which were active at the time the OutOfMemoryError was thrown, including any inner scopes. For further information about using the **-Xdump** option, see “Using the -Xdump option” on page 159.

The following example shows some typical output. All the values are output as hexadecimal values.

```
NULL ------------------------------------------------------------------------
0SECTION MEMINFO subcomponent dump routine
NULL ================================
NULL
1STMEMTYPE Object Memory
NULL region start end size name
1STHEAP 0x080AF72C 0xF65D4000 0xF69D0000 0x003FC000 Default
NULL
1STMEMUSAGE Total memory available: 04194304 (0x00400000)
1STMEMUSAGE Total memory in use: 03711279 (0x0038A12F)
1STMEMUSAGE Total memory free: 00483025 (0x00075ED1)
NULL
NULL region start end size name
1STHEAP 0x080AF74C 0xF53FF008 0xF63FF008 0x01000000 Immortal
NULL
1STMEMUSAGE Total memory available: 16777216 (0x01000000)
1STMEMUSAGE Total memory in use: 00463288 (0x000711B8)
1STMEMUSAGE Total memory free: 16313928 (0x00F8EE48)
NULL
1STSEGTYPE Internal Memory
NULL segment start alloc end type size
1STSEGMENT 0xF6475018 0xE8E70050 0xE8E70050 0xE8E80050 0x01000040 0x00010000
1STSEGMENT 0xF6474DD8 0xE9020028 0xE9020028 0xE9030028 0x01000040 0x00010000
<< lines removed for clarity >>

NULL
1STSEGUSAGE Total memory available: 00742340 (0x000B53C4)
1STSEGUSAGE Total memory in use: 00000000 (0x00000000)
1STSEGUSAGE Total memory free: 00742340 (0x000B53C4)
NULL
1STSEGTYPE Class Memory
NULL segment start alloc end type size
1STSEGMENT 0xF64AA7E0 0xF64E1900 0xF64E2AA0 0xF64E5900 0x00010040 0x00004008
1STSEGMENT 0xF64AA788 0xF64CF638 0xF64D58B0 0xF64DF638 0x00020040 0x00010000
<< lines removed for clarity >>

NULL
1STSEGUSAGE Total memory available: 02123876 (0x00206864)
1STSEGUSAGE Total memory in use: 01911828 (0x001D2C14)
1STSEGUSAGE Total memory free: 00212048 (0x00033C50)
NULL
1STSEGTYPE JIT Code Cache
NULL segment start alloc end type size
1STSEGMENT 0x0810AFA8 0xED3FE000 0xF53FE000 0xF53FE000 0x00000068 0x08000020
NULL
1STSEGUSAGE Total memory available: 134217760 (0x08000020)
1STSEGUSAGE Total memory in use: 134217728 (0x08000000)
1STSEGUSAGE Total memory free: 00000032 (0x00000020)
NULL
1STSEGTYPE JIT Data Cache
NULL segment start alloc end type size
1STSEGMENT 0x0810B0E0 0xE92FC008 0xE9308064 0xED2FC008 0x00000048 0x04000000
NULL
1STSEGUSAGE Total memory available: 67108864 (0x04000000)
1STSEGUSAGE Total memory in use: 00049244 (0x0000C05C)
1STSEGUSAGE Total memory free: 67059620 (0x03FF3FA4)
```

Locks, monitors, and deadlocks (LOCKS)

An example of the LOCKS component part of a Javadump taken during a deadlock.

A lock, also referred to as a monitor, prevents more than one entity from accessing a shared resource. Each object in Java has an associated lock, obtained by using a synchronized block or method. In the case of the JVM, threads compete for various resources in the JVM and locks on Java objects.

This example was taken from a deadlock test program where two threads "DeadLockThread 0" and "DeadLockThread 1" were unsuccessfully attempting to synchronize (Java keyword) on two java/lang/Integers.

You can see in the example (highlighted) that "DeadLockThread 1" has locked the object instance java/lang/Integer@004B2290. The monitor has been created as a result of a Java code fragment looking like "synchronize(count0)", and this monitor has "DeadLockThread 1" waiting to get a lock on this same object instance (count0 from the code fragment). Below the highlighted section is another monitor locked by "DeadLockThread 0" that has "DeadLockThread 1" waiting.

This classic deadlock situation is caused by an error in application design; Javadump is a major tool in the detection of such events.

```
------------------------------------------------------------------------
LOCKS subcomponent dump routine
==============================

   Monitor pool info:
    Current total number of monitors: 2

Monitor Pool Dump (flat & inflated object-monitors):
      sys_mon_t:0x00039B40 infl_mon_t: 0x00039B80:
       java/lang/Integer@004B22A0/004B22AC: Flat locked by "DeadLockThread 1"
                                            (0x41DAB100), entry count 1
            Waiting to enter:
                "DeadLockThread 0" (0x41DAAD00)      sys_mon_t:0x00039B98 infl_mon_t: 0x00039BD8:
      java/lang/Integer@004B2290/004B229C: Flat locked by "DeadLockThread 0"
                                                          (0x41DAAD00), entry count 1
            Waiting to enter:
                "DeadLockThread 1" (0x41DAB100)
JVM System Monitor Dump (registered monitors):
          Thread global lock (0x00034878): <unowned>
          NLS hash table lock (0x00034928): <unowned>
          portLibrary_j9sig_async_monitor lock (0x00034980): <unowned>
          Hook Interface lock (0x000349D8): <unowned>
            < lines removed for brevity >

======================
Deadlock detected !!!
---------------------

  Thread "DeadLockThread 1" (0x41DAB100)
    is waiting for:
      sys_mon_t:0x00039B98 infl_mon_t: 0x00039BD8:
      java/lang/Integer@004B2290/004B229C:
    which is owned by:
  Thread "DeadLockThread 0" (0x41DAAD00)
    which is waiting for:
      sys_mon_t:0x00039B40 infl_mon_t: 0x00039B80:
      java/lang/Integer@004B22A0/004B22AC:
    which is owned by:
  Thread "DeadLockThread 1" (0x41DAB100)
```

Threads and stack trace (THREADS)

For the application programmer, one of the most useful pieces of a Java dump is the THREADS section. This section shows a complete list of Java threads and stack traces.

A thread is alive if it has been started but not yet stopped. A Java thread is implemented by a native thread of the operating system. Each thread is represented by a line such as:

```
"Signal Dispatcher" TID:0x41509200, j9thread_t:0x0003659C, state:R,prio=5
       (native thread ID:5820, native priority:0, native policy:SCHED_OTHER)
    at com/ibm/misc/SignalDispatcher.waitForSignal(Native Method)
    at com/ibm/misc/SignalDispatcher.run(SignalDispatcher.java:84)
```

From WebSphere Real Time for RT Linux V2 SR3, Java thread names are visible in the operating system when using the ps command. For further information about using the ps command, see "Examining process information" on page 128.

A Java dump that is produced from a no-heap real-time thread could have some missing information. For threads where the thread name object is not visible from the no-heap real-time thread, the text "(access error)" is printed instead of the actual thread name.

The properties on the first line are thread name, identifier, JVM data structure address, current state, and Java priority. The properties on the second line are the native operating system thread ID, native operating system thread priority and native operating system scheduling policy.

From WebSphere Real Time for RT Linux releases SR2 to SR3, several threads have been assigned new identifying names. These names are visible in three ways:

- Appearing in javacore files. Not all threads appear in javacore files.
- When listing threads from the O/S using the ps command.
- When using the java.lang.Thread.getName() method.

The following table provides more information about new or affected thread names.

Table 17. New thread names in WebSphere Real Time for RT Linux

Detail of thread	Old thread name	New thread name
An internal JVM thread used by the garbage collection module to dispatch the finalization of objects by secondary threads.	`main`	`Finalizer master`
The alarm thread used by the garbage collector.	`Metronome GC Alarm Thread`	`GC Alarm`
The slave threads used for garbage collection.	`Gc Slave Thread`	`Gc Slave`
An internal JVM thread used by the just-in-time compiler module to sample the usage of methods in the application.		`JIT Sampler`
A thread used by the VM to manage signals received by the application, whether externally or internally generated.		`Signal Reporter`

The default names of real-time threads (javax.realtime.RealtimeThread) created in Java code have acquired new names as well. The names of these threads have been shortened from `RealtimeThread-x` to `RTThread-x`, where "x" is the thread number.

No-heap real-time thread names have been altered in the same way. Their names have changed from `NoHeapRealtimeThread-x` to `NHRTThread-x`.

The Java thread priority is mapped to an operating system priority value in a platform-dependent manner. A large value for the Java thread priority means that the thread has a high priority. In other words, the thread runs more frequently than lower priority threads. For further details of how this works for Java threads, real-time threads and no-heap real-time threads, see "Priority mapping and inheritance" on page 63.

The values of state can be:

- R - Runnable - the thread is able to run when given the chance.
- CW - Condition Wait - the thread is waiting. For example, because:
  - A sleep() call is made
  - The thread has been blocked for I/O
  - A wait() method is called to wait on a monitor being notified
  - The thread is synchronizing with another thread with a join() call
- S – Suspended – the thread has been suspended by another thread.
- Z – Zombie – the thread has been killed.
- P – Parked – the thread has been parked by the new concurrency API (java.util.concurrent).
- B – Blocked – the thread is waiting to obtain a lock that something else currently owns.

Understanding Java thread details:

Below each Java thread is a stack trace, which represents the hierarchy of Java method calls made by the thread.

The following example is taken from the same Javadump that is used in the `LOCKS` example. Two threads, "DeadLockThread 0" and "DeadLockThread 1", are in blocked state. The application code path that resulted in the deadlock between "DeadLockThread 0" and "DeadLockThread 1" can clearly be seen.

There is no current thread because all the threads in the application are blocked. A user signal generated the Javadump.

```
------------------------------------------------------------------------
THREADS subcomponent dump routine
================================

Current Thread Details
----------------------

All Thread Details
------------------

Full thread dump J9SE VM (J2RE 5.0 IBM J9 2.3 Linux x86-32 build 20060714_07194_lHdSMR,
native threads):
    "DestroyJavaVM helper thread" TID:0x41508A00, j9thread_t:0x00035EAC, state:CW, prio=5
           (native thread ID:3924, native priority:0, native policy:SCHED_OTHER, scope:00A6D068)
    "JIT Compilation Thread" TID:0x41508E00, j9thread_t:0x000360FC, state:CW, prio=11
           (native thread ID:188, native priority:11, native policy:SCHED_OTHER, scope:00A6D068)
    "Signal Dispatcher" TID:0x41509200, j9thread_t:0x0003659C, state:R, prio=5
           (native thread ID:3192, native priority:0, native policy:SCHED_OTHER, scope:00A6D084)
        at com/ibm/misc/SignalDispatcher.waitForSignal(Native Method)
        at com/ibm/misc/SignalDispatcher.run(SignalDispatcher.java:84)
    "DeadLockThread 0" TID:0x41DAAD00, j9thread_t:0x42238A1C, state:B, prio=5
```

```
        (native thread ID:1852, native priority:0, native policy:SCHED_OTHER, scope:00A6D068)
    at Test$DeadlockThread0.SyncMethod(Test.java:112)
    at Test$DeadlockThread0.run(Test.java:131)
"DeadLockThread 1" TID:0x41DAB100, j9thread_t:0x42238C6C, state:B, prio=5
        (native thread ID:1848, native priority:0, native policy:SCHED_OTHER, scope:00A6D068)
    at Test$DeadlockThread1.SyncMethod(Test.java:160)
    at Test$DeadlockThread1.run(Test.java:141)
```

Current Thread Details section

If the Javadump is triggered on a running Java thread, the `Current Thread Details` section shows a Java thread name, properties and stack trace. This output is generated if, for example, a GPF occurs on a Java thread, or if the com.ibm.jvm.Dump.JavaDump() API is called.

```
Current Thread Details
----------------------
   "main" TID:0x0018D000, j9thread_t:0x002954CC, state:R, prio=5
           (native thread ID:0xAD0, native priority:0x5, native policy:UNKNOWN)
        at com/ibm/jvm/Dump.JavaDumpImpl(Native Method)
        at com/ibm/jvm/Dump.JavaDump(Dump.java:20)
        at Test.main(Test.java:26)
```

Typically, Javadumps triggered by a user signal do not show a current thread because the signal is handled on a native thread, and the Java threads are suspended while the Javadump is produced.

Stack backtrace

The stack backtrace provides a full stack trace of the failing native thread. You can use the stack backtrace to determine if a crash is caused by an error in the JVM or the native application.

At the top of the stack trace are the JVM signal handler and dump handling routines. The actual point of failure is this stack frame: `/home/sdk/jre/lib/i386/realtime/libj9trc25.so [0xb743a71a]`.

```
Native thread id:0x000007DF
/home/sdk/jre/lib/i386/realtime/libj9prt25.so [0xb76a5086]
/home/sdk/jre/lib/i386/realtime/libj9dmp25.so [0xb745ad20]
/home/sdk/jre/lib/i386/realtime/libj9dmp25.so [0xb745994e]
/home/sdk/jre/lib/i386/realtime/libj9dmp25.so [0xb745f4e7]
/home/sdk/jre/lib/i386/realtime/libj9dmp25.so [0xb74508e2]
/home/sdk/jre/lib/i386/realtime/libj9dmp25.so [0xb74534d8]
/home/sdk/jre/lib/i386/realtime/libj9prt25.so [0xb76ad454]
/home/sdk/jre/lib/i386/realtime/libj9dmp25.so [0xb74534a5]
/home/sdk/jre/lib/i386/realtime/libj9dmp25.so [0xb7460424]
/home/sdk/jre/lib/i386/realtime/libj9vm25.so [0xb76d1926]
/home/sdk/jre/lib/i386/realtime/libj9prt25.so [0xb76ad454]
/home/sdk/jre/lib/i386/realtime/libj9vm25.so [0xb76d0e58]
/home/sdk/jre/lib/i386/realtime/libj9prt25.so [0xb76adf68]
[0xffffe440]
/home/sdk/jre/lib/i386/realtime/libj9trc25.so [0xb743a71a]
/home/sdk/jre/lib/i386/realtime/libj9trc25.so [0xb743a964]
/home/sdk/jre/lib/i386/realtime/libj9trc25.so [0xb74376a5]
/home/sdk/jre/lib/i386/realtime/libj9hookable25.so [0xb7f66950]
/home/sdk/jre/lib/i386/realtime/libj9vm25.so [0xb76d6b24]
/home/sdk/jre/lib/i386/realtime/libj9vm25.so [0xb76d8b9d]
/home/sdk/jre/lib/i386/realtime/libjclscar_25.so [0xb6dc648d]
/home/sdk/jre/lib/i386/realtime/libjclscar_25.so [0xb6e0dc18]
/home/sdk/jre/lib/i386/realtime/libjclscar_25.so [0xb6e12ce1]
/home/sdk/jre/lib/i386/realtime/libjclscar_25.so(J9VMDllMain+0xf0) [0xb6e12dee]
/home/sdk/jre/lib/i386/realtime/libj9vm25.so [0xb76f8ac2]
/home/sdk/jre/lib/i386/realtime/libj9vm25.so [0xb771d360]
/home/sdk/jre/lib/i386/realtime/libj9vm25.so [0xb76f898e]
```

```
/home/sdk/jre/lib/i386/realtime/libj9vm25.so [0xb76fd229]
/home/sdk/jre/lib/i386/realtime/libj9prt25.so [0xb76ad454]
/home/sdk/jre/lib/i386/realtime/libj9vm25.so [0xb76f55bc]
/home/sdk/jre/lib/i386/realtime/libj9vm25.so(JNI_CreateJavaVM+0x99) [0xb76e55b9]
/home/sdk/jre/lib/i386/realtime/libjvm.so(JNI_CreateJavaVM+0xb0a) [0xb773db64]
/home/sdk/jre/lib/i386/j9vm/libjvm.so(JNI_CreateJavaVM+0x24e) [0xb7f74fb6]
sdk/jre/bin/java [0x804b5cc]
sdk/jre/bin/java(JavaMain+0x7e) [0x8049d0e]
/lib/tls/libpthread.so.0 [0x4ec9f7f1]
/lib/tls/libc.so.6(__clone+0x5e) [0x4eb3171e]
```

Shared Classes (SHARED CLASSES)

An example of the shared classes section that includes summary information about the shared data cache.

```
SHARED CLASSES subcomponent dump routine
========================================

Cache Summary
-----------------

    ROMClass start address     = 0xE4EFD000
    ROMClass end address       = 0xE55FD000
    Metadata start address     = 0xE55FD778
    Cache end address          = 0xE5600000
    Runtime flags              = 0x34368297
    Cache generation           = 5

    Cache size                 = 7356040
    Free bytes                 = 1011412
    ROMClass bytes             = 2628000
    AOT bytes                  = 4151573368
    Java Object bytes          = 0
    ReadWrite bytes            = 3720
    Byte data bytes            = 92
    Metadata bytes             = 147106744

    Number ROMClasses          = 651
    Number AOT Methods         = 1691
    Number Java Objects        = 0
    Number Classpaths          = 2
    Number URLs                = 0
    Number Tokens              = 0
    Number Stale classes       = 0
    Percent Stale classes      = 0%

    Cache is 86% full

Cache Memory Status
-----------------
    Cache Name                    Memory type             Cache path

    sharedcc_rtjaxxon             Memory mapped file      /tmp/javasharedresources/C250D2A32P_sha

Cache Lock Status
-----------------
    Lock Name                     Lock type               TID owning lock

    Cache write lock              File lock               Unowned
    Cache read/write lock         File lock               Unowned
```

Classloaders and Classes (CLASSES)

An example of the classloader (CLASSES) section that includes Classloader summaries and Classloader loaded classes. Classloader summaries are the defined class loaders and the relationship between them. Classloader loaded classes are the classes that are loaded by each classloader.

See the Diagnostics Guide for information about the parent-delegation model.

In this example, there are the standard three classloaders:

- Application classloader (sun/misc/Launcher$AppClassLoader), which is a child of the extension classloader.
- The Extension classloader (sun/misc/Launcher$ExtClassLoader), which is a child of the bootstrap classloader.
- The Bootstrap classloader. Also known as the System classloader.

The example that follows shows this relationship. Take the application classloader with the full name sun/misc/Launcher$AppClassLoader. Under `Classloader summaries`, it has flags `-----ta-`, which show that the class loader is t=trusted and a=application (See the example for information on class loader flags). It gives the number of loaded classes (1) and the parent classloader as sun/misc/Launcher$ExtClassLoader.

Under the `ClassLoader loaded classes` heading, you can see that the application classloader has loaded three classes, one called Test at address 0x41E6CFE0.

In this example, the System class loader has loaded a large number of classes, which provide the basic set from which all applications derive.

```
------------------------------------------------------------------------
CLASSES subcomponent dump routine
================================
Classloader summaries
        12345678: 1=primordial,2=extension,3=shareable,4=middleware,
                  5=system,6=trusted,7=application,8=delegating
        p---st--        Loader *System*(0x00439130)
                        Number of loaded libraries 5
                        Number of loaded classes 306
                        Number of shared classes 306
        -x--st--        Loader sun/misc/Launcher$ExtClassLoader(0x004799E8),
                        Parent *none*(0x00000000)
                        Number of loaded classes 0
        -----ta-        Loader sun/misc/Launcher$AppClassLoader(0x00484AD8),
                        Parent sun/misc/Launcher$ExtClassLoader(0x004799E8)
                        Number of loaded classes 1
ClassLoader loaded classes
Loader *System*(0x00439130)
  java/security/CodeSource(0x41DA00A8)
  java/security/PermissionCollection(0x41DA0690)
    << 301 classes removed for clarity >>
  java/util/AbstractMap(0x4155A8C0)
  java/io/OutputStream(0x4155ACB8)
  java/io/FilterOutputStream(0x4155AE70)
Loader sun/misc/Launcher$ExtClassLoader(0x004799E8)
Loader sun/misc/Launcher$AppClassLoader(0x00484AD8)
  Test(0x41E6CFE0)
  Test$DeadlockThread0(0x41E6D410)
  Test$DeadlockThread1(0x41E6D6E0)
```

Environment variables and Javadump

Although the preferred mechanism of controlling the production of Javadumps is now by the use of dump agents using **-Xdump:java**, you can also use the previous mechanism, environment variables.

The following table details environment variables specifically concerned with Javadump production:

Environment Variable	Usage Information
DISABLE_JAVADUMP	Setting **DISABLE_JAVADUMP** to true is the equivalent of using **–Xdump:java:none** and stops the default production of javadumps.
IBM_JAVACOREDIR	The default location into which the Javacore will be written.
JAVA_DUMP_OPTS	Use this environment variable to control the conditions under which Javadumps (and other dumps) are produced. See “Dump agent environment variables” on page 172 for more information.
IBM_JAVADUMP_OUTOFMEMORY	By setting this environment variable to false, you disable Javadumps for an out-of-memory exception.

Using Heapdump

The term Heapdump describes the IBM Virtual Machine for Java mechanism that generates a dump of all the live objects that are on the Java heap; that is, those that are being used by the running Java application.

This dump is stored in a Portable Heap Dump (PHD) file, a compressed binary format. You can use various tools on the Heapdump output to analyze the composition of the objects on the heap and (for example) help to find the objects that are controlling large amounts of memory on the Java heap and the reason why the Garbage Collector cannot collect them.

This chapter describes:

Related information

“Heapdumps” on page 164
Heapdumps produce phd format files by default.

Getting Heapdumps

By default, a Heapdump is produced when the Java heap is exhausted. Heapdumps can be generated in other situations by use of **-Xdump:heap**.

See “Using dump agents” on page 159 for more detailed information about generating dumps based on specific events. Heapdumps can also be generated programmatically by use of the com.ibm.jvm.Dump.HeapDump() method from inside the application code.

To see which events will trigger a dump, use **-Xdump:what**. See “Using dump agents” on page 159 for more information.

By default, Heapdumps contain information about all the objects in the JVM's memory areas, Heap, Immortal and Scoped memory. To produce individual Heapdumps for each memory area, see “Enabling multiple Heapdumps for real-time JVMs.”

By default, Heapdumps are produced in PHD format. To produce Heapdumps in text format, see “Enabling text formatted ("classic") Heapdumps.”

Environment variables can also affect the generation of Heapdumps (although this is a deprecated mechanism). See “Environment variables and Heapdump” on page 189 for more details.

Enabling multiple Heapdumps for real-time JVMs

The generated Heapdump is by default a single file containing information about all Java objects in all memory areas, Heap memory, Immortal memory and Scoped memory. The main reason to produce multiple dumps is so that each individual heap area can be analyzed using the traditional Heapdump tools without modification.

About this task

You can obtain separate Heapdumps containing information about Java objects in each memory area by using the **request=multiple** option with **-Xdump:heap**. Note that you must repeat the default settings of the request option as well, so you need to specify `request=multiple+exclusive+prepwalk+compact`. This produces a set of Heapdumps with an extra field in the name indicating the specific memory area:

```
heapdump.%id.%Y%m%d.%H%M%S.%pid.phd
```

where *%id* identifies the Heapdump file as containing objects in Heap memory, Immortal memory, or a specific area of Scoped memory.

There are 4 types of heap represented by the following names: “Default”, “Immortal”, “Scope” and “Other”. The Heapdump code replaces the %id in the heap label with one of these names concatenated with an identifier (typically numeric), for example: `heapdump.Immortal12994208.20060807.093653.7684.txt.`

Example

```
java -Xrealtime -Xdump:heap:defaults:request=multiple+exclusive+compact+prepwalk <java program>
```

Using this extra option produces multiple Heapdumps in the portable Heapdump (phd) format.

```
java -Xrealtime -Xdump:heap:defaults:request=multiple+exclusive+compact+prepwalk,opts=CLASSIC <java
```

Using this extra option produces multiple Heapdumps in the CLASSIC text format.

Using the -Xdump:what option will print out the dump agents on JVM startup, and is useful for checking the dump options that are in place.

Related information

“Available tools for processing Heapdumps” on page 189
There are several tools available for Heapdump analysis through IBM support Web sites.

Enabling text formatted ("classic") Heapdumps

The generated Heapdump is by default in the binary, platform-independent, PHD format, which can be examined using the available tooling.

For more information, see "Available tools for processing Heapdumps." However, an immediately readable view of the heap is sometimes useful. You can obtain this view by using the **opts=** suboption with **-Xdump:heap** (see "Using dump agents" on page 159). For example:

- **-Xdump:heap:opts=CLASSIC** will start the default Heapdump agents using classic rather than PHD output.
- **-Xdump:heap:defaults:opts=CLASSIC+PHD** will enable both classic and PHD output by default for all Heapdump agents.

You can also define one of the following environment variables:

- **IBM_JAVA_HEAPDUMP_TEST**, which allows you to perform the equivalent of **opts=PHD+CLASSIC**
- **IBM_JAVA_HEAPDUMP_TEXT**, which allows the equivalent of **opts=CLASSIC**

Available tools for processing Heapdumps

There are several tools available for Heapdump analysis through IBM support Web sites.

The preferred Heapdump analysis tool is the IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer. The tool is available in IBM Support Assistant: http://www.ibm.com/software/support/isa/. Information about the tool can be found at http://www.ibm.com/developerworks/java/jdk/tools/memoryanalyzer/

Further details of the range of available tools can be found at http://www.ibm.com/support/docview.wss?uid=swg24009436

Related tasks

"Enabling multiple Heapdumps for real-time JVMs" on page 188
The generated Heapdump is by default a single file containing information about all Java objects in all memory areas, Heap memory, Immortal memory and Scoped memory. The main reason to produce multiple dumps is so that each individual heap area can be analyzed using the traditional Heapdump tools without modification.

Using -Xverbose:gc to obtain heap information

Use the **-Xverbose:gc** utility to obtain information about the Java Object heap in real time while running your Java applications.

To activate this utility, run Java with the **-verbose:gc** option:

```
java -verbose:gc
```

For more information, see "Using verbose:gc information" on page 51.

Environment variables and Heapdump

Although the preferred mechanism for controlling the production of Heapdumps is now the use of dump agents with **-Xdump:heap**, you can also use the previous mechanism, environment variables.

The following table details environment variables specifically concerned with Heapdump production:

Environment Variable	Usage Information
IBM_HEAPDUMP **IBM_HEAP_DUMP**	Setting either of these to any value (such as true) enables heap dump production by means of signals.
IBM_HEAPDUMPDIR	The default location into which the Heapdump will be written.
JAVA_DUMP_OPTS	Use this environment variable to control the conditions under which Heapdumps (and other dumps) are produced. See "Dump agent environment variables" on page 172 for more information .
IBM_HEAPDUMP_OUTOFMEMORY	By setting this environment variable to false, you disable Heapdumps for an OutOfMemory condition.
IBM_JAVA_HEAPDUMP_TEST	Use this environment variable to cause the JVM to generate both phd and text versions of Heapdumps. Equivalent to **opts=PHD+CLASSIC** on the **-Xdump:heap** option.
IBM_JAVA_HEAPDUMP_TEXT	Use this environment variable to cause the JVM to generate a text (human readable) Heapdump. Equivalent to **opts=CLASSIC** on the **-Xdump:heap** option.

Text (classic) Heapdump file format

The text or classic Heapdump is a list of all object instances in the heap, including object type, size, and references between objects, in a human-readable format.

Header record

The header record is a single record containing a string of version information.

```
// Version: <version string containing SDK level, platform and JVM build level>
```

Example:

```
// Version: J2RE 6.0 IBM J9 2.5 Linux x86-32 build 20081016_024574_lHdRSr
```

Object records

Object records are multiple records, one for each object instance on the heap, providing object address, size, type, and references from the object.

```
<object address, in hexadecimal> [<length in bytes of object instance, in decimal>]
OBJ <object type> <class block reference, in hexadecimal>
<heap reference, in hexadecimal <heap reference, in hexadecimal> ...
```

The object address and heap references are in the heap, but the class block address is outside the heap. All references found in the object instance are listed, including those that are null values. The object type is either a class name including package or a primitive array or class array type, shown by its standard JVM type signature, see "Java VM type signatures" on page 192. Object records can also contain additional class block references, typically in the case of reflection class instances.

Examples:

An object instance, length 28 bytes, of type java/lang/String:

```
0x00436E90 [28] OBJ java/lang/String
```

A class block address of java/lang/String, followed by a reference to a char array instance:

```
0x415319D8 0x00436EB0
```

An object instance, length 44 bytes, of type char array:

```
0x00436EB0 [44] OBJ [C
```

A class block address of char array:

```
0x41530F20
```

An object of type array of java/util/Hashtable Entry inner class:

```
0x004380C0 [108] OBJ [Ljava/util/Hashtable$Entry;
```

An object of type java/util/Hashtable Entry inner class:

```
0x4158CD80 0x00000000 0x00000000 0x00000000 0x00000000 0x00421660 0x004381C0
0x00438130 0x00438160 0x00421618 0x00421690 0x00000000 0x00000000 0x00000000
0x00438178 0x004381A8 0x004381F0 0x00000000 0x004381D8 0x00000000 0x00438190
0x00000000 0x004216A8 0x00000000 0x00438130 [24] OBJ java/util/Hashtable$Entry
```

A class block address and heap references, including null references:

```
0x4158CB88 0x004219B8 0x004341F0 0x00000000
```

Class records

Class records are multiple records, one for each loaded class, providing class block address, size, type, and references from the class.

```
<class block address, in hexadecimal> [<length in bytes of class block, in decimal>]
CLS <class type>
<class block reference, in hexadecimal> <class block reference, in hexadecimal> ...
<heap reference, in hexadecimal> <heap reference, in hexadecimal>...
```

The class block address and class block references are outside the heap, but the class record can also contain references into the heap, typically for static class data members. All references found in the class block are listed, including those that are null values. The class type is either a class name including package or a primitive array or class array type, shown by its standard JVM type signature, see"Java VM type signatures" on page 192.

Examples:

A class block, length 32 bytes, for class java/lang/Runnable:

```
0x41532E68 [32] CLS java/lang/Runnable
```

References to other class blocks and heap references, including null references:

```
0x4152F018 0x41532E68 0x00000000 0x00000000 0x00499790
```

A class block, length 168 bytes, for class java/lang/Math:

```
0x00000000 0x004206A8 0x00420720 0x00420740 0x00420760 0x00420780 0x004207B0
0x00421208 0x00421270 0x00421290 0x004212B0 0x004213C8 0x00421458 0x00421478
0x00000000 0x41589DE0 0x00000000 0x4158B340 0x00000000 0x00000000 0x00000000
0x4158ACE8 0x00000000 0x4152F018 0x00000000 0x00000000 0x00000000
```

Trailer record 1

Trailer record 1 is a single record containing record counts.

```
// Breakdown - Classes: <class record count, in decimal>,
Objects: <object record count, in decimal>,
ObjectArrays: <object array record count, in decimal>,
PrimitiveArrays: <primitive array record count, in decimal>
```

Example:

```
// Breakdown - Classes: 321, Objects: 3718, ObjectArrays: 169,
PrimitiveArrays: 2141
```

Trailer record 2

Trailer record 2 is a single record containing totals.

```
// EOF:  Total 'Objects',Refs(null) :
<total object count, in decimal>,
<total reference count, in decimal>
(,total null reference count, in decimal>)
```

Example:

```
// EOF:  Total 'Objects',Refs(null) : 6349,23240(7282)
```

Java VM type signatures

The Java VM type signatures are abbreviations of the Java types are shown in the following table:

Java VM type signatures	Java type
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L <*fully qualified-class*> ;	<*fully qualified-class*>
[<*type*>	<*type*>[ ] (array of <*type*>)
(<*arg-types*>) <*ret-type*>	method

Using system dumps and the dump viewer

The JVM can generate native system dumps, also known as core dumps, under configurable conditions. System dumps are typically quite large. Use the gdb tool to analyze a system dump on Linux.

Dump agents are the primary method for controlling the generation of system dumps. See “Using dump agents” on page 159 for more information. To maintain backwards compatibility, the JVM supports the use of environment variables for system dump triggering. See “Dump agent environment variables” on page 172 for more information.

This chapter tells you about system dumps and how to use the dump viewer. It contains these topics:

- "Overview of system dumps"
- "System dump defaults"
- "Using the dump viewer" on page 194

Related information

"System dumps" on page 163
System dumps involve dumping the address space and as such are generally very large.

"Debugging with gdb" on page 131
The GNU debugger (gdb) allows you to examine the internals of another program while the program executes or retrospectively to see what a program was doing at the moment that it crashed.

Overview of system dumps

The JVM can produce system dumps in response to specific events. A system dump is a raw binary dump of the process memory when the dump agent is triggered by a failure or by an event for which a dump is requested.

Generally, you use a tool to examine the contents of a system dump. A dump viewer tool is provided in the SDK, as described in this section, or you could use a platform-specific debugger, such as gdb, to examine the dump. For dumps triggered by a General Protection Fault (GPF), dumps produced by the JVM contain some context information that you can read. You can find this failure context information by searching in the dump for the eye-catcher

```
J9Generic_Signal_Number
```

For example:

```
J9Generic_Signal_Number=00000004 ExceptionCode=c0000005 ExceptionAddress=7FAB506D ContextFlags=000
Handler1=7FEF79C0 Handler2=7FED8CF0 InaccessibleAddress=0000001C
EDI=41FEC3F0 ESI=00000000 EAX=41FB0E60 EBX=41EE6C01
ECX=41C5F9C0 EDX=41FB0E60
EIP=7FAB506D ESP=41C5F948 EBP=41EE6CA4
Module=/mysdk/sdk/jre/lib/i386/realtime/libj9jit25.so
Module_base_address=7F8D0000 Offset_in_DLL=001e506d

Method_being_compiled=org/junit/runner/JUnitCore.runMain([Ljava/lang/String;)Lorg/junit/runner/Res
```

Dump agents are the primary method for controlling the generation of system dumps. See "Using dump agents" on page 159 for more information on dump agents.

System dump defaults

There are default agents for producing system dumps when using the JVM.

Using the **-Xdump:what** option shows the following system dump agent:

```
-Xdump:system:
    events=gpf+abort,
    label=/home/user/core.%Y%m%d.%H%M%S.%pid.dmp,
    range=1..0,
    priority=999,
    request=serial
```

This output shows that by default a system dump is produced in these cases:

- A general protection fault occurs. (For example, branching to memory location 0, or a protection exception.)
- An abort is encountered. (For example, native code has called abort() or when using `kill -ABRT` on Linux)

Attention: The JVM used to produce this output when a SIGSEGV signal was encountered. This behavior is no longer supported. Use the ABRT signal to produce dumps.

Using the dump viewer

System dumps are produced in a platform-specific binary format, typically as a raw memory image of the process that was running at the time the dump was initiated. The SDK dump viewer allows you to navigate around the dump, and obtain information in a readable form, with symbolic (source code) data where possible.

You can view Java information (for example, threads and objects on the heap) and native information (for example, native stacks, libraries, and raw memory locations).

Dump extractor: jextract

To use the dump viewer you must first use the jextract tool on the system dump. The jextract tool obtains platform specific information such as word size, endianness, data structure layouts, and symbolic information. It puts this information into an XML file. jextract also collects other useful files, depending on the platform, including trace files and copies of executable files and libraries and, by default, compresses these into a single `.zip` file for use in subsequent problem diagnosis.

The jextract tool must be run in the same mode (-Xrealtime or not) and the same JVM level (ideally the same machine) that was being used when the dump was produced. The combination of the dump file and the XML file produced by jextract allows the dump viewer (jdmpview) to analyze and display Java information.

The extent to which jextract can analyze the information in a dump is affected by the state of the JVM when it was taken. For example, the dump could have been taken while the JVM was in an inconsistent state. The exclusive and prepwalk dump options ensure that the JVM (and the Java heap) is in a safe state before taking a system dump:

```
-Xdump:system:defaults:request=exclusive+prepwalk
```

Setting this option adds a significant performance reduction when taking a system dump; which could cause problems in rare situations. This option is not enabled by default.

jextract is in the directory `sdk/jre/bin`.

To use jextract, enter the following at a command prompt:

```
jextract -Xrealtime  <core_file> [<zip_file>]
```

or

```
jextract -Xrealtime  -nozip <core_file> [<xml_file>]
```

The jextract tool accepts these parameters:

-help
Provides usage information.

-nozip
Do not compress the output data.

By default, output is written to a file called <*core_file*>.zip in the current directory. This file is a compressed file that contains:

- The dump
- XML produced from the dump, containing details of useful internal JVM information
- Other files that can help in diagnosing the dump (such as trace entry definition files)

You can use the jdmpview tool to analyze the extracted dump locally.

If you run jextract on a JVM level that is different from the one for which the dump was produced you will see the following messages:

```
J9RAS.buildID is incorrect (found e8801ed67d21c6be, expecting eb4173107d21c673).
This version of jextract is incompatible with this dump.
Failure detected during jextract, see previous message(s).
```

This error message also occurs if you use jextract without the **-Xrealtime** flag when you are processing a core that was produced with the **-Xrealtime** flag. Similarly you must not use the **-Xrealtime** flag with jextract when processing a core that was produced by running Java in the non-real-time mode.

The contents of the .zip file produced and the contents of the XML are subject to change, you are advised not to design tools based on the contents of these.

For system dumps generated on some recent levels of Linux you might see the following error messages when you run jextract:

```
<!--extracting monitors-->
[ERR] could not read byte at F7F502EF
...
```

These errors mean that jextract was unable to find all the expected information about JVM monitors. You can still use the dump viewer to examine the dump, but you might see further error messages when accessing JVM monitor information.

Dump viewer: jdmpview

The dump viewer is a tool that allows you to examine the contents of system dumps produced from the JVM. The dump viewer requires metadata created by the jextract tool. It allows you to view both Java and native information from the time the dump was produced.

jdmpview is in the directory sdk/bin.

To start jdmpview, from a shell prompt, enter:

```
jdmpview -Xrealtime  -zip <zip file>
```

or

```
jdmpview -Xrealtime  -core <core file> [-xml <xml file>]
```

The jdmpview -Xrealtime tool accepts these parameters:

-core *<core file>*
Specify a dump file.

-xml *<xml file>*
Specify a metadata file. jdmpview will guess the name of the XML file if the **-xml** option is not present.

-zip *<zip file>*
Specify a .zip file containing the core file and associated XML file (produced by jextract).

Note: The **-core** and **-xml** options can be used with the **-zip** option to specify the core and XML files in the compressed file. Without the **-core** or **-xml** options, jdmpview will guess the names of the files in the compressed file.

After jdmpview -Xrealtime processes the arguments with which it was launched, it displays this message:

```
For a list of commands, type "help"; for how to use "help", type "help help"
>
```

When you see this message, you can start using commands.

When jdmpview is used with the **-zip** option, temporary disk space is needed to uncompress the dump files from the compressed file. jdmpview will use the system temporary directory, `/tmp` on Linux. An alternative temporary directory can be specified using the Java system property **java.io.tmpdir**. jdmpview will display an error message if insufficient disk space is available in the temporary directory.

You can significantly improve the performance of jdmpview against large dumps by ensuring that your system has enough memory available to avoid paging. On large dumps (that is, ones with large numbers of objects on the heap), you might have to run jdmpview using the **-Xmx** option to increase the maximum heap available:

```
jdmpview -Xrealtime  -J-Xmx<n> -zip <zip file>
```

To pass command-line arguments to the JVM, you must prefix them with **-J**.

Problems to tackle with the dump viewer

Dumps of JVM processes can arise either when you use the **-Xdump** option on the command line or when the JVM is not in control (such as user-initiated dumps).

The extent to which jextract can analyze the information in a dump is affected by the state of the JVM when it was taken. For example, the dump could have been taken while the JVM was in an inconsistent state. The exclusive and prepwalk dump options ensure that the JVM (and the Java heap) is in a safe state before taking a system dump:

```
-Xdump:system:defaults:request=exclusive+prepwalk
```

Setting this option adds a significant performance reduction when taking a system dump; which could cause problems in rare situations. This option is not enabled by default.

jdmpview is most useful in diagnosing customer-type problems and problems with the class libraries. A typical scenario is OutOfMemoryError exceptions in customer applications.

For problems involving gpfs, ABENDS, SIGSEVs, and similar problems, you will obtain more information by using a system debugger (gdb) with the dump file. The syntax for the gdb command is

```
gdb <full_java_path> <system_dump_file>
```

For example:

```
gdb /sdk/jre/bin/java core.20060808.173312.9702.dmp
```

jdmpview can still provide useful information when used alone. Because jdmpview allows you to observe stacks and objects, the tool enables introspection into a Java program in the same way as a Java debugger. It allows you to examine objects, follow reference chains and observe Java stack contents. The main difference (other than the user interface) is that the program state is frozen; thus no stepping can occur. However, this allows you to take periodic program snapshots and perform analysis to see what is happening at different times.

Commands for use with jdmpview

jdmpview -Xrealtime is an interactive, command-line tool to explore the information from a JVM system dump and perform various complex analysis functions.

cd <*directory_name*>

Changes the current working directory, used for log files. Changes the current working directory to <*directory_name*>, checking to see if it exists and is a directory before making the change. The log files can be found in the current working directory; a change to the current working directory has no effect on the current log file setting because the logging filename is converted to an absolute path when set. Note: to see what the current working directory is set to, use the **pwd** command.

deadlock

Displays information about deadlocks if there are any set. This command shows detailed information about deadlocks or "no deadlocks detected" if there are no deadlocks. A deadlock situation consists of one or more deadlock loops and zero or more branches attached to those loops. This command prints out each branch attached to a loop and then the loop itself. If there is a split in a deadlock branch, separate branches are created for each side of the split in the branch. Deadlock branches start with a monitor that has no threads waiting on it and the continues until it reaches a monitor that exists in another deadlock branch or loop. Deadlock loops start and end with the same monitor.

Monitors are represented by their owner and the object associated with the given monitor. For example, the **3435 (0x45ae67)** output represents the monitor that is owned by the thread with id 3435 and is associated the object at address 0x45ae67. Objects can be viewed by using a command like **x/j 0x45ae67** and threads can be viewed using a command like **info thread 3435**.

find
<*pattern*>,<*start_address*>,<*end_address*>,<*memory_boundary*>,<*bytes_to_print*>,<*matches_to_display*>

This command searches for <*pattern*> in the memory segment from <*start_address*> to <*end_address*> (both inclusive), and outputs the first <*matches_to_display*> matching addresses. It also displays the next <*bytes_to_print*> bytes for the last match.

By default, the **find** command searches for the supplied pattern at every byte in the range. If you know the pattern is aligned to a particular byte boundary, you can specify <*memory_boundary*> to search once every <*memory_boundary*> bytes, for example, 4 or 8 bytes.

findnext
Finds the next instance of the last string passed to **find**. This command is used in conjunction with **find** or **findptr** command to continue searching for the next matches. It repeats the previous **find** or **findptr** command (depending on which command is most recently issued) starting from the last match.

findptr
<*pattern*>,<*start_address*>,<*end_address*>,<*memory_boundary*>,<*bytes_to_print*>,<*matches_to_display*>
Searches memory for the given pointer. **findptr** searches for <*pattern*> as a pointer in the memory segment from <*start_address*> to <*end_address*> (both inclusive), and outputs the first <*matches_to_display*> matching addresses that start at the corresponding <*memory_boundary*>. It also displays the next <*bytes_to_print*> bytes for the last match.

help [<*command_name*>]
Displays a list of commands or help for a specific command. With no parameters, **help** displays the complete list of commands currently supported. When a <*command_name*> is specified, **help** lists that command's sub-commands if it has sub-commands; otherwise, the command's complete description is displayed.

info thread [*|<*thread_name*>]
Displays information about Java and native threads. The command prints the following information about the current thread (no arguments), all threads ("*"), or the specified thread:
- Thread id
- Registers
- Stack sections
- Thread frames: procedure name and base pointer
- Associated Java thread (if applicable):
  - Name of Java thread
  - Address of associated java.lang.Thread object
  - State according to JVMTI specification
  - State relative to java.lang.Thread.State
  - The monitor the thread is waiting to enter or waiting on notify
  - Thread frames: base pointer, method, and filename:line

info system
Displays the following information about the system the core dump:
- amount of memory
- operating system
- virtual machine(s) present

info class [<*class_name*>]
Displays the inheritance chain and other data for a given class. If no parameters are passed to **info class**, it prints the number of instances of each class and the total size of all instances of each class as well as the total number of instances of all classes and the total size of all objects. if a class name is passed to **info class**, it prints the following information about that class:
- name
- ID
- superclass ID
- class loader ID
- modifiers
- number of instances and total size of instances
- inheritance chain
- fields with modifiers (and values for static fields)
- methods with modifiers

info proc
Displays threads, command line arguments, environment variables, and shared modules of current process.

Note: To view the shared modules used by a process, use the **info sym** command.

info jitm
Displays JIT and AOT compiled methods and their addresses:
- Method name and signature
- Method start address
- Method end address

info lock
Displays a list of available monitors and locked objects

info sym
Displays a list of available modules. For each process in the address spaces, this command outputs a list of module sections for each module with their start and end addresses, names, and sizes.

info mmap
Displays a list of all memory segments in the address space: start address and size.

info heap [* | <*heap_name*>**]**
Using no arguments displays the heap names and heap sections.

Using either "*" or a heap name displays the following information about all heaps or the specified heap:
- heap name
- (heap size and occupancy)
- heap sections
  - section name
  - section size
  - whether the section is shared
  - whether the section is executable
  - whether the section is read only

hexdump <*hex_address*> <*bytes_to_print*>
Displays a section of memory in a hexdump-like format. Displays <*bytes_to_print*> bytes of memory contents starting from <*hex_address*>.

+ Displays the next section of memory in hexdump-like format. This command is used in conjunction with the hexdump command to allow easy scrolling forwards through memory. It repeats the previous hexdump command starting from the end of the previous one.

- Displays the previous section of memory in hexdump-like format. This command is used in conjunction with the hexdump command to allow easy scrolling backwards through memory. It repeats the previous hexdump command starting from a position before the previous one.

pwd
Displays the current working directory which is the directory where log files are stored.

quit
Exits the core file viewing tool; any log files that are currently open are closed before exit.

set logging <*options*>
Configures logging settings, starts logging, or stops logging. This allows the results of commands to be logged to a file.

The options are:

[on | off]
Turns logging on or off. (Default: off)

file <*filename*>
sets the file to log to; this will be relative to the directory returned by the pwd command unless an absolute path is specified; if the file is set while logging is on, the change will take effect the next time logging is started. Not set by default.

overwrite [on | off]
Turns overwriting of the specified log file on or off. When overwrite is off, log messages will be appended to the log file. When overwrite is on, the log file will be overwritten after the **set logging** command. (Default: off)

redirect [on | off]
Turns redirecting to file on or off (on means that non-error output goes only to the log file when logging is on, off means that non-error output goes to both the console and the log file); redirection must be turned off logging can be turned off. (Default: off)

show logging
Displays the current logging settings:
- set_logging = [on | off]
- set_logging_file =
- set_logging_overwrite = [on | off]
- set_logging_redirect = [on | off]
- current_logging_file = - file that is currently being logged to; it could be different from set_logging_file, if that value was changed after logging was started.

whatis <*hex_address*>
Displays information about what is stored at the given memory address, <*hex_address*>. This command examines the memory location at <*hex_address*> and tries to find out more information about this address. For example, whether it is in an object in a heap or in the byte codes associated with a class method.

x/ (examine)
Passes number of items to display and unit size ('b' for byte (8 bit), 'h' for half word (16 bit), 'w' for word (32 bit), 'g' for giant word (64 bit)) to sub-command (for example x/12bd). This is similar to the use of the **x/** command in gdb (including use of defaults).

x/J [<*0xaddr*> | <*class_name*>**]**
Displays information about a particular object or all objects of a class. If given class name, all static fields with their values are printed, followed by all objects of that class with their fields and values. If given an object address (in hex), static fields for that object's class are not printed; the other fields and values of that object are printed along with its address.

Note: This command ignores the number of items and unit size passed to it by the **x/** command.

x/D *<0xaddr>*
Displays the integer at the specified address, adjusted for the endianness of the architecture this dump file is from.

Note: This command uses the number of items and unit size passed to it by the **x/** command.

x/X *<0xaddr>*
Displays the hex value of the bytes at the specified address, adjusted for the endianness of the architecture this dump file is from.

Note: This command uses the number of items and unit size passed to it by the **x/** command.

x/K *<0xaddr>*
Displays the value of each section (where the size is defined by the pointer size of this architecture) of memory, adjusted for the endianness of the architecture this dump file is from, starting at the specified address. It also displays a module with a module section and an offset from the start of that module section in memory if the pointer points to that module section. If no symbol is found, it displays a "*" and an offset from the current address if the pointer points to an address in 4KB (4096 bytes) of the current address. While this command can work on an arbitrary section of memory, it is probably most useful when used on a section of memory that refers to a stack frame. To find the memory section of a thread's stack frame, use the info thread command.

Note: This command uses the number of items and unit size passed to it by the **x/** command.

Example session

This example session illustrates a selection of the commands available and their use.

In the example session, some lines have been removed for clarity (and terseness). Some comments (contained inside braces) are included to explain various aspects with some comments on individual lines looking like:

```
{ comment }
```

User input is prefaced by a ">".

```
{First, invoke DTFJView using the jdmpview launcher, passing in the name of a dump.}

> jdmpview -Xrealtime-core core.20081020.145957.32289.0001.dmp
DTFJView version 1.0.22, using DTFJ API version 1.3
Loading image from DTFJ...
For a list of commands, type "help"; for how to use "help", type "help help"


{the output produced by help is illustrated below}

>help

        info
         thread   displays information about Java and native threads
         system   displays information about the system the core dump is from
         class    prints inheritance chain and other data for a given class
         proc     displays threads, command line arguments, environment variables,
                                                            and shared modules of current process
         jitm     displays JIT'ed methods and their addresses
         ls       outputs a list of available monitors and locked objects
         sym      outputs a list of available modules
         mmap     outputs a list of all memory segments in the address space
         heap     displays information about Java heaps
        hexdump   outputs a section of memory in a hexdump-like format
        +         displays the next section of memory in hexdump-like format
```

```
        -         displays the previous section of memory in hexdump-like format
        find      searches memory for a given string
        deadlock  displays information about deadlocks if there are any
        set
         logging  configures several logging-related parameters, starts/stops logging
        show
         logging  displays the current values of logging settings
        quit      exits the core file viewing tool
        whatis    gives information about what is stored at the given memory address
        cd        changes the current working directory, used for log files
        pwd       displays the current working directory
        findnext  finds the next instance of the last string passed to "find"
        findptr   searches memory for the given pointer
        help      displays list of commands or help for a specific command
        x/        works like "x/" in gdb (including use of defaults): passes number of items to display
                       and unit size ('b' for byte, 'h' for halfword, 'w' for word, 'g' for giant word)
                       to sub-command (ie. x/12bd)
         j        displays information about a particular object or all objects of a class
         d        displays the integer at the specified address
         x        displays the hex value of the bytes at the specified address
         k        displays the specified memory section as if it were a stack frame



{In jdmpview setting an output file could be done from the invocation, in DTFJView it must be
                                                                  done using the "set logging" command. }

> set logging file log.txt

        log file set to "log.txt"

> set logging on

        logging turned on; outputting to "/home/test/log.txt"

> show logging

        set_logging = on
        set_logging_file = "log.txt"
        set_logging_overwrite = off
        set_logging_redirect = off

        current_logging_file = "/home/test/log.txt"



>info thread   << Displays info on current thread. Use "info thread *" for information on all threads.


        native threads for address space # 0
         process id: 28836

          thread id: 28836
           registers:
            cs       = 0x00000073   ds       = 0x0000007b   eax      = 0x00000000   ebp      = 0xbfe32064
            ebx      = 0xb7e9e484   ecx      = 0x00000000   edi      = 0xbfe3245c   edx      = 0x00000002
            efl      = 0x00010296   eip      = 0xb7e89120   es       = 0xc010007b   esi      = 0xbfe32471
            esp      = 0xbfe31c2c   fs       = 0x00000000   gs       = 0x00000033   ss       = 0x0000007b
           stack sections:
            0xbfe1f000 to 0xbfe34000 (length 0x15000)
           stack frames:
            bp: 0xbfe32064   proc name: /home/test/sdk/jre/bin/java::_fini

 ==== lines removed for terseness====
 ==== lines removed for terseness====

            bp: 0x00000000   proc name: <unknown location>
           properties:

           associated Java thread: <no associated Java thread>



>info system

System: Linux
System Memory: 2323206144 bytes
```

```
Virtual Machine(s):
Runtime #1:
Java(TM) SE Runtime Environment(build jvmxi3260rt-20081016_24574)
IBM J9 VM(J2RE 1.6.0 IBM J9 2.5 Linux x86-32 jvmxi3260rt-20081016_24574 (JIT enabled, AOT enabled)
J9VM - 20081016_024574_lHdRSr
JIT - r10_20081015_2000
GC - 20081016_AA)


> info class  << Information on all classes

 Runtime #1:
        instances      total size  class name
                0               0  java/util/regex/Pattern$Slice
                0               0  java/lang/Byte
                0               0  java/lang/CharacterDataLatin1
                2              96  sun/nio/cs/StreamEncoder$ConverterSE
             1015           36540  java/util/TreeMap$Entry

 ==== lines removed for terseness====
 ==== lines removed for terseness====

                 2              48  [java/io/File
                5             104  [java/io/ObjectStreamField
                0               0  java/lang/StackTraceElement

  Total number of objects: 9240
  Total size of objects: 562618



> info class java/util/Random  << Information on a specific class

 Runtime #1:
name = java/util/Random

        ID = 0x81c9fb0    superID = 0x80ea450
        classLoader = 0x82307e8    modifers: public synchronized

        number of instances:     1
        total size of instances: 32 bytes

Inheritance chain....

        java/lang/Object
           java/util/Random

Fields......

          static fields for "java/util/Random"
           static final long serialVersionUID = 3905348978240129619 (0x363296344bf00a53)
           private static final long multiplier = 25214903917 (0x5deece66d)
           private static final long addend = 11 (0xb)

 ==== lines removed for terseness====

          non-static fields for "java/util/Random"
           private long seed
           private double nextNextGaussian
           private boolean haveNextNextGaussian


Methods......

Bytecode range(s): 81fb41c -- 81fb430:  public void <init>()

 ==== lines removed for terseness====

Bytecode range(s): 81fb624 -- 81fb688:  public synchronized double nextGaussian()
Bytecode range(s): 81fb69c -- 81fb6a4:  static void <clinit>()


> info proc

        address space # 0

         Thread information for current process:
          Thread id: 28836
```

```
         Command line arguments used for current process:
          /home/test/sdk/jre/bin/java -Xmx100m -Xms100m -Xtrace: DeadlockCreator

         Environment variables for current process:
          IBM_JAVA_COMMAND_LINE=/home/test/sdk/jre/bin/java -Xmx100m -Xms100m -Xtrace: DeadlockCreator
          LIBPATH=.:/home/test/sdk/jre/bin:/home/test/sdk/jre/bin/j9vm:/usr/bin/gcc:
          HISTSIZE=1000

 ==== lines removed for terseness====

          PATH=.:/home/test/sdk/bin:/home/test/sdk/jre/bin/j9vm:/home/test/sdk/jre/bin:
                /usr/bin/gcc:/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/test/bin
          TERM=xterm



> info jitm

        start=0xb11e241c  end=0xb11e288f   DeadlockCreator::main([Ljava/lang/String;)V
        start=0xb11e28bc  end=0xb11e64ca   DeadlockThreadA::syncMethod(LDeadlockThreadA;)V
        start=0xb11e64fc  end=0xb11ea0fa   DeadlockThreadB::syncMethod(LDeadlockThreadB;)V
        start=0xb11dd55c  end=0xb11dd570   java/lang/Object::<init>()V

   ==== lines removed for terseness====
 ==== lines removed for terseness====

        start=0xb11e0f54  end=0xb11e103e   java/util/zip/ZipEntry::initFields(J)V
        start=0xb11e0854  end=0xb11e0956   java/util/zip/Inflater::inflateBytes([BII)I
        start=0xb11e13d4  end=0xb11e14bf   java/util/zip/Inflater::reset(J)V

> info ls
(un-named monitor @0x835ae50 for object @0x835ae50)
 owner thread's id = <data unavailable>
 object = 0x835ae50

(un-named monitor @0x835b138 for object @0x835b138)
 owner thread's id = <data unavailable>
 object = 0x835b138

Thread global
Raw monitor: id = <unavailable>

NLS hash table
Raw monitor: id = <unavailable>

portLibrary_j9sig_sync_monitor
Raw monitor: id = <unavailable>

portLibrary_j9sig_asynch_reporter_shutdown_monitor
Raw monitor: id = <unavailable>


 ==== lines removed for terseness====
 ==== lines removed for terseness====

Thread public flags mutex
Raw monitor: id = <unavailable>

Thread public flags mutex
Raw monitor: id = <unavailable>

JIT-QueueSlotMonitor-21
Raw monitor: id = <unavailable>

Locked objects...
 java/lang/Class@0x835ae50 is locked by a thread with id <data unavailable>
 java/lang/Class@0x835b138 is locked by a thread with id <data unavailable>


> info sym

        modules for address space # 0
         process id: 28836



> info mmap
Address: 0x1000   size: 0x1000 (4096)
Address: 0x8048000   size: 0xd000 (53248)
```

```
Address: 0x8055000   size: 0x2000 (8192)
Address: 0x8057000   size: 0x411000 (4263936)

 ==== lines removed for terseness====
 ==== lines removed for terseness====

Address: 0xffffe460   size: 0x18 (24)
Address: 0xffffe478   size: 0x24 (36)
Address: 0xffffe5a8   size: 0x78 (120)


> info heap object <<Displays information on the object heap, "info heap *" displays
                                                                                  information on all heaps.

        Runtime #1
         Heap #1:  Default
          Size of heap: 104857600 bytes
          Occupancy :   562618 bytes  (0.53%)
          Section #1:  Contiguous heap extent at 0xb1439000 (0x6400000 bytes)
           Size:        104857600 bytes
           Shared:      false
           Executable:  false
           Read Only:   false

> hexdump 0xb1439000 200

b1439000: 70da0e08 0e800064 00000000 00000000  |p......d........|
b1439010: d0c20e08 05800464 00000000 80000000  |.......d........|
b1439020: 00000100 02000300 04000500 06000700  |................|
b1439030: 08000900 0a000b00 0c000d00 0e000f00  |................|
b1439040: 10001100 12001300 14001500 16001700  |................|
b1439050: 18001900 1a001b00 1c001d00 1e001f00  |................|
b1439060: 20002100 22002300 24002500 26002700  | .!."#.$.%.&.'.|
b1439070: 28002900 2a002b00 2c002d00 2e002f00  |(.).*.+.'.-.../.|
b1439080: 30003100 32003300 34003500 36003700  |0.1.2.3.4.5.6.7.|
b1439090: 38003900 3a003b00 3c003d00 3e003f00  |8.9.:.;.<.=.>.?.|
b14390a0: 40004100 42004300 44004500 46004700  |@.A.B.C.D.E.F.G.|
b14390b0: 48004900 4a004b00 4c004d00 4e004f00  |H.I.J.K.L.M.N.O.|
b14390c0: 50005100 52005300                    |P.Q.R.S.

> +

b14390c8: 54005500 56005700 58005900 5a005b00  |T.U.V.W.X.Y.Z.[.|
b14390d8: 5c005d00 5e005f00 60006100 62006300  |\.].^._.`.a.b.c.|
b14390e8: 64006500 66006700 68006900 6a006b00  |d.e.f.g.h.i.j.k.|
b14390f8: 6c006d00 6e006f00 70007100 72007300  |l.m.n.o.p.q.r.s.|
b1439108: 74007500 76007700 78007900 7a007b00  |t.u.v.w.x.y.z.{.|
b1439118: 7c007d00 7e007f00 e0df0e08 01804864  ||.}.~.........Hd|
b1439128: 00000000 00000000 90e00e08 01804c64  |..............Ld|
b1439138: 00000000 00000000 78f30e08 0e805064  |........x.....Pd|
b1439148: 00000000 b89b43b1 b89b43b1 00000000  |......C...C.....|
b1439158: 78f30e08 0e805664 00000000 109c43b1  |x.....Vd......C.|
b1439168: 109c43b1 00000000 78f30e08 0e805c64  |..C.....x.....\d|
b1439178: 00000000 709c43b1 709c43b1 00000000  |....p.C.p.C.....|
b1439188: 78f30e08 0e806264                    |x.....bd

> -

b1439000: 70da0e08 0e800064 00000000 00000000  |p......d........|
b1439010: d0c20e08 05800464 00000000 80000000  |.......d........|
b1439020: 00000100 02000300 04000500 06000700  |................|
b1439030: 08000900 0a000b00 0c000d00 0e000f00  |................|
b1439040: 10001100 12001300 14001500 16001700  |................|
b1439050: 18001900 1a001b00 1c001d00 1e001f00  |................|
b1439060: 20002100 22002300 24002500 26002700  | .!."#.$.%.&.'.|
b1439070: 28002900 2a002b00 2c002d00 2e002f00  |(.).*.+.'.-.../.|
b1439080: 30003100 32003300 34003500 36003700  |0.1.2.3.4.5.6.7.|
b1439090: 38003900 3a003b00 3c003d00 3e003f00  |8.9.:.;.<.=.>.?.|
b14390a0: 40004100 42004300 44004500 46004700  |@.A.B.C.D.E.F.G.|
b14390b0: 48004900 4a004b00 4c004d00 4e004f00  |H.I.J.K.L.M.N.O.|
b14390c0: 50005100 52005300                    |P.Q.R.S.


> whatis 0xb143a000

 Runtime #1:
         heap #1 - name: object heap

0xb143a000 is within the heap segment: b1439000 -- b7839000
```

```
0xb143a000 is within an object on the heap.
        Offset 8 within [char instance @ 0xb1439ff8


{ find command parameters are: <pattern>,<start_address>,<end_address>,<memory_boundary>,
                                                        <bytes_to_print>,<matches_to_display> }

> find a,0b1439000,0xb1440000,10,20,5
#0: 0xb1439c00
#1: 0xb1439c46
#2: 0xb143a1be
#3: 0xb143a1c8
#4: 0xb143a1e6

b143a1e6: 61007000 70006500 6e006900 6e006700  |a.p.p.e.n.i.n.g.|
b143a1f6: 45007800                             |E.x.

> findnext <<Repeats find command, starting from last match.
#0: 0xb143a72c
#1: 0xb143b3f2
#2: 0xb143b47e
#3: 0xb143b492
#4: 0xb143b51e

b143b51e: 61002e00 73007000 65006300 69006600  |a...s.p.e.c.i.f.|
b143b52e: 69006300                             |i.c.

> findnext
#0: 0xb143b532
#1: 0xb143b5e6
#2: 0xb143b5fa
#3: 0xb143b71c
#4: 0xb143bac8

b143bac8: 61007000 00000000 10cd0e08 0e80b46e  |a.p...........n|
b143bad8: 00000000                             |....


{ x/j can be passed an object address or a class name }
> x/j 0xb1439000

 Runtime #1:
        heap #1 - name: object heap

         java/lang/String$CaseInsensitiveComparator @ 0xb1439000

{If passed an object address the (non-static) fields and values of the object will be printed }

> x/j java/lang/Float

 Runtime #1:
        heap #1 - name: object heap

          static fields for "java/lang/Float"
           public static final float POSITIVE_INFINITY = Infinity (0x7f800000)
           public static final float NEGATIVE_INFINITY = -Infinity (0xffffffffff800000)
           public static final float NaN = NaN (0x7fc00000)
           public static final float MAX_VALUE = 3.4028235E38 (0x7f7fffff)
           public static final float MIN_VALUE = 1.4E-45 (0x1)
           public static final int SIZE = 32 (0x20)
           public static final Class TYPE = <object> @ 0x80ec368
           private static final long serialVersionUID = -2671257302660747028 (0xdaedc9a2db3cf0ec)

          <no object of class "java/lang/Float" exists>

{If passed a class name the static fields and their values are printed, followed by all objects of
                                                                                    that class }


> x/d 0xb1439000

        0xb1439000: 135191152 <<Integer at specified address


> x/x 0xb1439000

        0xb1439000: 080eda70 <<Hex value ofthe bytes at specified address
```

```
{ "cd" and "pwd" are self explanatory. }
> pwd
        /home/test

> cd deadlock/
> pwd

        /home/test/deadlock

> quit
```

jdmpview commands quick reference

A short list of the commands you use with jdmpview.

The following table shows the jdmpview - quick reference:

Command	Sub-command	Description
help		Displays a list of commands or help for a specific command.
info		
	thread	Displays information about Java and native threads.
	system	Displays information about the system the core dump is from.
	class	Displays the inheritance chain and other data for a given class.
	proc	Displays threads, command line arguments, environment variables, and shared modules of current process.
	jitm	Displays JIT and AOT compiled methods and their addresses.
	lock	Displays a list of available monitors and locked objects.
	sym	Displays a list of available modules.
	mmap	Displays a list of all memory segments in the address space.
	heap	Displays information about all heaps or the specified heap.
hexdump		Displays a section of memory in a hexdump-like format.
+		Displays the next section of memory in hexdump-like format.
-		Displays the previous section of memory in hexdump-like format.
whatis		Displays information about what is stored at the given memory address.
find		Searches memory for a given string.
findnext		Finds the next instance of the last string passed to “find”.
findptr		Searches memory for the given pointer.
x/ (examine)		Examine works like "x/" in gdb (including use of defaults): passes number of items to display and unit size ('b' for byte (8 bit), 'h' for half word (16 bit), 'w' for word (32 bit), 'g' for giant word (64 bit)) to sub-command (for example x/12bd).
	J	Displays information about a particular object or all objects of a class.
	D	Displays the integer at the specified address.

Command	Sub-command	Description
	X	Displays the hex value of the bytes at the specified address.
	K	Displays the specified memory section as if it were a stack frame.
deadlock		Displays information about deadlocks if there are any set.
set logging		Configures logging settings, starts logging, or stops logging. This allows the results of commands to be logged to a file.
show logging		Displays the current values of logging settings.
cd		Changes the current working directory, used for log files.
pwd		Displays the current working directory.
quit		Exits exits the core file viewing tool; any log files that are currently open will be closed before the tool exits.

Tracing Java applications and the JVM

JVM trace is a trace facility that is provided in all IBM-supplied JVMs with minimal affect on performance. In most cases, the trace data is kept in a compact binary format, that can be formatted with the Java formatter that is supplied.

Tracing is enabled by default, together with a small set of trace points going to memory buffers. You can enable tracepoints at runtime by using levels, components, group names, or individual tracepoint identifiers.

This chapter describes JVM trace in:

- "What can be traced?"
- "Default tracing" on page 210
- "Where does the data go?" on page 211
- "Controlling the trace" on page 212
- "Determining the tracepoint ID of a tracepoint" on page 230
- "Application trace" on page 231
- "Using method trace" on page 234

Trace is a powerful tool to help you diagnose the JVM.

Related concepts

"Troubleshooting the Metronome Garbage Collector" on page 51
Using the command-line options, you can control the frequency of Metronome garbage collection, out of memory exceptions, and the Metronome behavior on explicit system calls.

What can be traced?

You can trace JVM internals, applications, and Java method or any combination of those.

JVM internals

The IBM Virtual Machine for Java is extensively instrumented with tracepoints for trace. Interpretation of this trace data requires knowledge of the internal operation of the JVM, and is provided to diagnose JVM problems.

No guarantee is given that tracepoints will not vary from release to release and from platform to platform.

Applications
JVM trace contains an application trace facility that allows tracepoints to be placed in Java code to provide trace data that will be combined with the other forms of trace. There is an API in the com.ibm.jvm.Trace class to support this. Note that an instrumented Java application runs only on an IBM-supplied JVM.

Java methods
You can trace entry to and exit from Java methods run by the JVM. You can select method trace by classname, method name, or both. You can use wildcards to create complex method selections.

JVM trace can produce large amounts of data in a very short time. Before running trace, think carefully about what information you need to solve the problem. In many cases, where you need only the trace information that is produced shortly before the problem occurs, consider using the **wrap** option. In many cases, just use internal trace with an increased buffer size and snap the trace when the problem occurs. If the problem results in a thread stack dump or operating system signal or exception, trace buffers are snapped automatically to a file that is in the current directory. The file is called: Snap*nnnn.yyyymmdd.hhmmssth.process*.trc.

You must also think carefully about which components need to be traced and what level of tracing is required. For example, if you are tracing a suspected shared classes problem, it might be enough to trace all components at level 1, and j9shr at level 9, while maximal can be used to show parameters and other information for the failing component.

Types of tracepoint

There are two types of tracepoints inside the JVM: regular and auxiliary.

Regular tracepoints

Regular tracepoints include:

- method tracepoints
- application tracepoints
- data tracepoints inside the JVM
- data tracepoints inside class libraries

You can display regular tracepoint data on the screen or save the data to a file. You can also use command line options to trigger specific actions when regular tracepoints fire. See the section “Detailed descriptions of trace options” on page 214 for more information about command line options.

Auxiliary tracepoints

Auxiliary tracepoints are a special type of tracepoint that can be fired only when another tracepoint is being processed. An example of auxiliary tracepoints are the tracepoints containing the stack frame information produced by the jstacktrace **-Xtrace:trigger** command. You cannot control where auxiliary tracepoint data is sent and you cannot set triggers on auxiliary tracepoints. Auxiliary tracepoint data is sent to the same destination as the tracepoint that caused them to be generated.

Default tracing

By default, the equivalent of the following trace command line is always available in the JVM:

```
-Xtrace:maximal=all{level1},exception=j9mm{gclogger}
```

The data generated by those tracepoints is continuously captured in wrapping, per thread memory buffers. (For information about specific options, see "Detailed descriptions of trace options" on page 214.)

You can find tracepoint output in the following diagnostics data:

- System dumps, extracted using jdmpview.
- Snap traces, generated when the JVM encounters a problem or an output file is specified. "Using dump agents" on page 159 describes more ways to create a snap trace.
- For exception trace only, in Javadumps.

Default memory management tracing

The default trace options are designed to ensure that Javadumps always contain a record of the most recent memory management history, regardless of how much JVM activity has occurred since the garbage collection cycle was last called.

The **exception=j9mm{gclogger}** clause of the default trace set specifies that a history of garbage collection cycles that have occurred in the JVM is continuously recorded. The gclogger group of tracepoints in the j9mm component constitutes a set of tracepoints that record a snapshot of each garbage collection cycle. These tracepoints are recorded in their own separate buffer called the exception buffer so that they are not overwritten by the higher frequency tracepoints of the JVM.

The **GC History** section of the Javadump is based on the information in the exception buffer. If a garbage collection cycle has occurred in a traced JVM the Javadump should contain a **GC History** section.

Default assertion tracing

The JVM includes assertions, implemented as special trace points. By default, internal assertions are detected and diagnostics logs are produced to help assess the error.

The JVM will continue executing after the logs have been produced, but assertion failures usually indicate a serious problem and the JVM might exit with a subsequent error. If the JVM does not encounter another error, an operator should still restart the JVM as soon as possible. A service request should be sent to IBM including the standard error output and the .trc and .dmp files produced.

When an assertion trace point is hit, a message similar to the following output is produced on the standard error stream:

```
16:43:48.671 0x10a4800    j9vm.209    *   ** ASSERTION FAILED ** at jniinv.c:251: ((javaVM == ((void *)0)))
```

This error stream is followed with information about the diagnostics logs produced:

```
JVMDUMP007I JVM Requesting System Dump using 'core.20060426.124348.976.dmp'
JVMDUMP010I System Dump written to core.20060426.124348.976.dmp
JVMDUMP007I JVM Requesting Snap Dump using 'Snap0001.20060426.124648.976.trc'
JVMDUMP010I Snap Dump written to Snap0001.20060426.124648.976.trc
```

Assertions are a special kind of trace point and can be enabled or disabled using the standard trace command-line options. See "Controlling the trace" on page 212 for more details.

Where does the data go?

Trace data can be written to a number of locations.

Trace data can go into:

- Memory buffers that can be dumped or snapped when a problem occurs
- One or more files that are using buffered I/O
- An external agent in real time
- stderr in real time
- Any combination of the above

Writing trace data to memory buffers

The use of memory buffers for trace is a very efficient method of running trace because no file I/O is performed until either a problem is detected or an API is used to snap the buffers to a file.

Buffers are allocated on a per-thread principle. This principle removes contention between threads and prevents trace data for individual threads from being swamped by other threads. For example, if one particular thread is not being dispatched, its trace information is still available when the buffers are dumped or snapped. Use **-Xtrace:buffers=<size>** to control the size of the buffer that is allocated to each thread.

Note: On some computers, power management affects the timers that trace uses, and gives misleading information. For reliable timing information, disable power management.

To examine the trace data captured in these memory buffers, you must snap or dump, then format the buffers.

Snapping buffers

Under default conditions, a running JVM collects a small amount of trace data in special wraparound buffers. This data is dumped to a snap file under the conditions listed as follows. You can use the **-Xdump:snap** option to vary the events that cause a snap trace to be produced. This file is in a binary format and requires the use of the supplied trace formatter so that you can read it.

Buffers are snapped when:

- An uncaught OutOfMemoryError occurs.
- An operating system signal or exception occurs.
- The com.ibm.jvm.Trace.snap() Java API is called.
- The JVMRI TraceSnap function is called.

The resulting snap file is placed into the current working directory with a name of the format Snap*nnnn.yyyymmdd.hhmmssth.process*.trc, where *nnnn* is a sequence number starting at 0001 (at JVM startup), *yyyymmdd* is the current date, *hhmmssth* is the current time, and *process* is the process identifier.

Extracting buffers from system dump

You can extract the buffers from a system dump core file by using the Dump Viewer.

Writing trace data to a file

You can write trace data to a file continuously as an extension to the in-storage trace, but, instead of one buffer per thread, at least two buffers per thread are allocated, and the data is written to the file before wrapping can occur.

This allocation allows the thread to continue to run while a full trace buffer is written to disk. Depending on trace volume, buffer size, and the bandwidth of the output device, multiple buffers might be allocated to a given thread to keep pace with trace data that is being generated.

A thread is never stopped to allow trace buffers to be written. If the rate of trace data generation greatly exceeds the speed of the output device, excessive memory usage might occur and cause out-of-memory conditions. To prevent this, use the **nodynamic** option of the **buffers** trace option. For long-running trace runs, a **wrap** option is available to limit the file to a given size. It is also possible to create a sequence of files when the trace output will move back to the first file once the sequence of files are full. See the **output** option for details. You must use the trace formatter to format trace data from the file.

Because trace data is buffered, if the JVM does not exit normally, residual trace buffers might not be flushed to the file. If the JVM encounters a fatal error, the buffers can be extracted from a system dump if that is available. When a snap file is created, all available buffers are always written to it.

External tracing

You can route trace to an agent by using JVMRI TraceRegister.

This mechanism allows a callback routine to be called immediately when any of the selected tracepoints is found without buffering the trace results. The trace data is in raw binary form. Further details can be found in the JVMRI section.

Tracing to stderr

For lower volume or non-performance-critical tracing, the trace data can be formatted and routed to stderr immediately without buffering.

For more information, see “Using method trace” on page 234.

Trace combinations

Most forms of trace can be combined, with the same or different trace data going to different destinations.

The exception to this is in-memory trace and trace to a file, which are mutually exclusive. When an ouput file is specified, any trace data that wraps in the in-memory case will be written to the file, and a new buffer given to the thread that filled its buffer. Without an output file specified, when a threads buffer is full, it wraps its trace data back to the beginning of its buffer.

Controlling the trace

You have several ways by which you can control the trace.

You can control the trace in several ways by using:

- The **-Xtrace** options when launching the JVM, including trace trigger events
- A trace properties file
- com.ibm.jvm.Trace API
- JVMTI and JVMRI from an external agent

Note:

1. The specification of trace options is cumulative. Multiple **-Xtrace** options are accepted on the command line and they are processed left to right order. Each one adds to the options set by the previous one (and to the default options), as if they had all been specified in one long comma-separated list in a single option. This cumulative specification is consistent with the related **-Xdump** option processing.
2. By default, trace options equivalent to the following are enabled:
   `-Xtrace:maximal=all{level1},exception=j9mm{gclogger}`
3. To disable the defaults (or any previous **-Xtrace** options), The **-Xtrace** keyword **none** also allows individual tracepoints or groups of tracepoints to be specified, like the other keywords. **none** is used in the same way to disable a set of tracepoints as **maximal**, **minimal** and the other options. However, instead of setting the maximal bit for a tracepoint, it will clear all previously set bits for that tracepoint. Thus **-Xtrace:none=all**
4. Many diagnostic tools start a JVM. When using the **IBM_JAVA_OPTIONS** environment variable trace to a file, starting a diagnostic tool might overwrite the trace data generated from your application. Use the command-line tracing options or add %d, %p or %t to the trace file name to prevent this from happening. See "Detailed descriptions of trace options" on page 214 for the appropriate trace option description.

Specifying trace options

The preferred way to control trace is through trace options that you specify either by using the **-Xtrace** option on the launcher command line. You can also use the **IBM_JAVA_OPTIONS** environment variable.

Some trace options have the form `<name>` and others are of the form `<name>=<value>`, where `<name>` is case-sensitive. Except where stated, `<value>` is not case-sensitive; the exceptions to this rule are filenames on some platforms, class names, and method names.

If an option value contains commas, it must be enclosed in braces. For example,

```
methods={java/lang/*,com/ibm/*}
```

Note that this applies only to options specified on the command line - not those specified in a properties file.

The syntax for specifying trace options depends on the launcher. Usually, it is:

```
java -Xrealtime -Xtrace:<name>,<another_name>=<value> HelloWorld
```

To switch off all tracepoints, use this option:

```
java -Xrealtime -Xtrace:none=all
```

If you specify other tracepoints without specifying **-Xtrace:none**, the tracepoints are added to the default set.

When you use the **IBM_JAVA_OPTIONS** environment variable, use this syntax:

```
set IBM_JAVA_OPTIONS=-Xtrace:<name>,<another_name>=<value>
```

or

```
export IBM_JAVA_OPTIONS=-Xtrace:<name>,<another_name>=<value>
```

If you use UNIX style shells, note that unwanted shell expansion might occur because of the characters used in the trace options. To avoid unpredictable results, enclose this command-line option in quotation marks. For example:

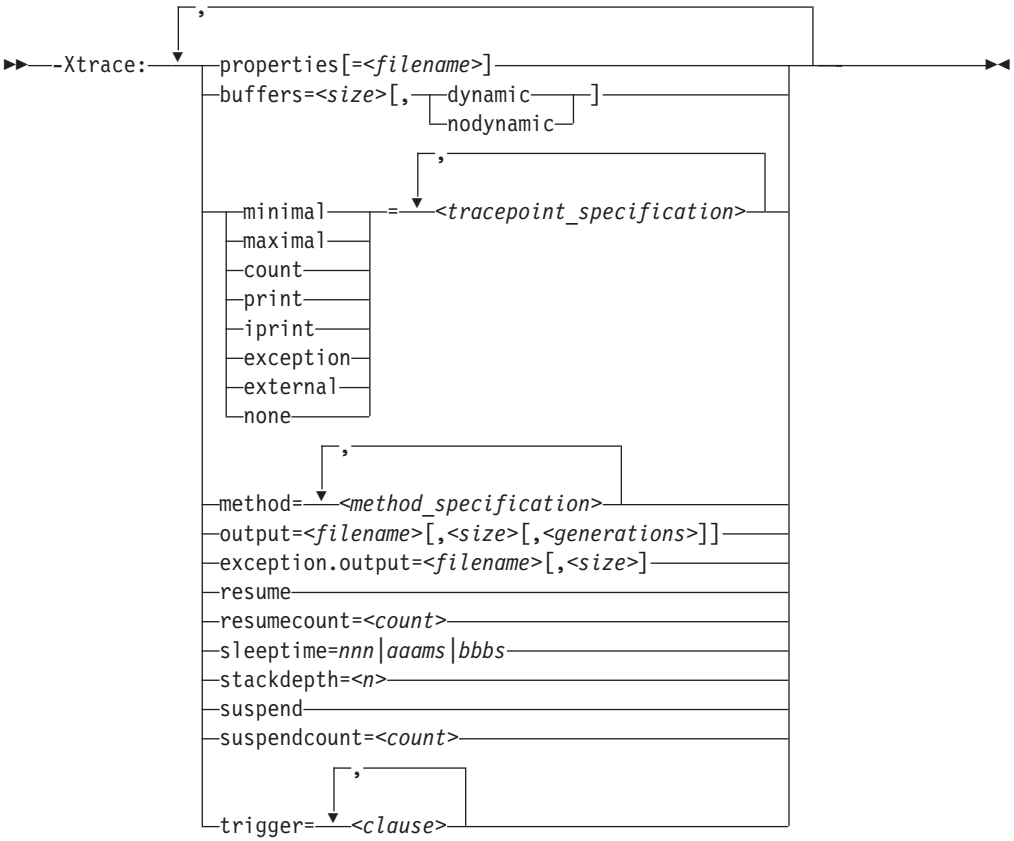
```
java -Xrealtime "-Xtrace:<name>,<another_name>=<value>" HelloWorld
```

For more information, see the manual for your shell.

Detailed descriptions of trace options

The options are processed in the sequence in which they are described here.

-Xtrace command-line option syntax



properties[=<*filename*>]:

You can use properties files to control trace. A properties file saves typing and, over time, causes a library of these files to be created. Each file is tailored to solving problems in a particular area.

This trace option allows you to specify in a file any of the other trace options, thereby reducing the length of the invocation command-line. The format of the file is a flat ASCII file that contains trace options. If <*filename*> is not specified, a default name of `IBMTRACE.properties` is searched for in the current directory. Nesting is not supported; that is, the file cannot contain a `properties` option. If any error is found when the file is accessed, JVM initialization fails with an explanatory error message and return code. All the options that are in the file are processed in the sequence in which they are stored in the file, before the next

option that is obtained through the normal mechanism is processed. Therefore, a command-line property always overrides a property that is in the file.

An existing restriction means that properties that take the form **<name>=<value>** cannot be left to default if they are specified in the property file; that is, you must specify a value, for example `maximal=all`.

Another restriction means that properties files are sensitive to white space. Do not add white space before, after, or within the trace options.

You can make comments as follows:

```
// This is a comment. Note that it starts in column 1
```

Examples

- Use `IBMTRACE.properties` in the current directory:

  ```
  -Xtrace:properties
  ```

  – Use `trace.prop` in the current directory:

    ```
    -Xtrace:properties=trace.prop
    ```

  – Use `c:\trc\gc\trace.props`:

    ```
    -Xtrace:properties=c:\trc\gc\trace.props
    ```

Here is an example property file:

```
minimal=all
// maximal=j9mm
maximal=j9shr
buffers=20k
output=c:\traces\classloader.trc
print=tpnid(j9vm.23-25)
```

buffers=dynamic | nodynamic:

You can specify how buffers are allocated when sending trace data to an output file.

From Java 6 SR 5, you can specify how buffers are allocated, without needing to specify the buffer size.

For more information about this option, see:
"buffers=*nnn*k | *nnn*m[,dynamic | nodynamic]"

buffers=*nnn*k | *nnn*m[,dynamic | nodynamic]:

You can modify the size of the buffers to change how much diagnostics output is provided in a snap dump. This buffer is allocated for each thread that makes trace entries.

From Java 6 SR 5, you do not need to specify the buffer size.

If external trace is enabled, the number of buffers is doubled; that is, each thread allocates two or more buffers. The same buffer size is used for state and exception tracing, but, in this case, buffers are allocated globally. The default is 8 KB per thread.

The **dynamic** and **nodynamic** options have meaning only when tracing to an output file. If **dynamic** is specified, buffers are allocated as needed to match the

rate of trace data generation to the output media. Conversely, if **nodynamic** is specified, a maximum of two buffers per thread is allocated. The default is **dynamic**. The dynamic option is effective only when you are tracing to an output file.

Note: If **nodynamic** is specified, you might lose trace data if the volume of trace data that is produced exceeds the bandwidth of the trace output file. Message `UTE115` is issued when the first trace entry is lost, and message `UTE018` is issued at JVM termination.

Examples

- Dynamic buffering with increased buffer size of 2 MB per thread:
  ```
  -Xtrace:buffers=2m
  ```
  or in a properties file:
  ```
  buffers=2m
  ```
- Trace buffers limited to two buffers per thread, each of 128 KB:
  ```
  -Xtrace:buffers={128k,nodynamic}
  ```
  or in a properties file:
  ```
  buffers=128k,nodynamic
  ```
- Trace using default buffer size of 8 KB, limited to two buffers per thread (Java 6 SR 5 or later):
  ```
  -Xtrace:buffers=nodynamic
  ```
  or in a properties file:
  ```
  buffers=nodynamic
  ```

Options that control tracepoint activation:

These options control which individual tracepoints are activated at runtime and the implicit destination of the trace data.

In some cases, you must use them with other options. For example, if you specify maximal or minimal tracepoints, the trace data is put into memory buffers. If you are going to send the data to a file, you must use an output option to specify the destination filename.

minimal=[!]<*tracepoint_specification*>[,...]
maximal=[!]<*tracepoint_specification*>[,...]
count=[!]<*tracepoint_specification*>[,...]
print=[!]<*tracepoint_specification*>[,...]
iprint=[!]<*tracepoint_specification*>[,...]
exception=[!]<*tracepoint_specification*>[,...]
external=[!]<*tracepoint_specification*>[,...]
none[=<*tracepoint_specification*>[,...]]

Note that all these properties are independent of each other and can be mixed and matched in any way that you choose.

From WebSphere Real Time for RT Linux V2 SR3, you must provide at least one tracepoint specification when using the **minimal**, **maximal**, **count**, **print**, **iprint**, **exception** and **external** options. In some older versions of the SDK the tracepoint specification defaults to 'all'.

Multiple statements of each type of trace are allowed and their effect is cumulative. To do this, you must use a trace properties file for multiple trace options of the same name.

minimal and maximal
minimal and **maximal** trace data is placed into internal trace buffers that can then be written to a snap file or written to the files that are specified in an output trace option. The **minimal** option records only the timestamp and tracepoint identifier. When the trace is formatted, missing trace data is replaced with the characters "???" in the output file. The **maximal** option specifies that all associated data is traced. If a tracepoint is activated by both trace options, **maximal** trace data is produced. Note that these types of trace are completely independent from any types that follow them. For example, if the **minimal** option is specified, it does not affect a later option such as **print**.

count
The **count** option requests that only a count of the selected tracepoints is kept. At JVM termination, all non-zero totals of tracepoints (sorted by tracepoint id) are written to a file, called utTrcCounters, in the current directory. This information is useful if you want to determine the overhead of particular tracepoints, but do not want to produce a large amount (GB) of trace data.

For example, to count the tracepoints used in the default trace configuration, use the following:

```
-Xtrace:count=all{level1},count=j9mm{gclogger}
```

print
The **print** option causes the specified tracepoints to be routed to stderr in real-time. The JVM tracepoints are formatted using `J9TraceFormat.dat`. The class library tracepoints are formatted by `TraceFormat.dat`. `J9TraceFormat.dat` and `TraceFormat.dat` are shipped in sdk/jre/lib and are automatically found by the runtime.

iprint
The **iprint** option is the same as the **print** option, but uses indenting to format the trace.

exception
When **exception** trace is enabled, the trace data is collected in internal buffers that are separate from the normal buffers. These internal buffers can then be written to a snap file or written to the file that is specified in an **exception.output** option.

The **exception** option allows low-volume tracing in buffers and files that are distinct from the higher-volume information that **minimal** and **maximal** tracing have provided. In most cases, this information is exception-type data, but you can use this option to capture any trace data that you want.

This form of tracing is channeled through a single set of buffers, as opposed to the buffer-per-thread approach for normal trace, and buffer contention might occur if high volumes of trace data are collected. A difference exists in the *<tracepoint_specification>* defaults for exception tracing; see "Tracepoint specification" on page 218.

Note: The exception trace buffers are intended for low-volume tracing. By default, the exception trace buffers log garbage collection event tracepoints, see "Default tracing" on page 210. You can send additional tracepoints to the exception buffers or switch off the garbage collection tracepoints. Changing the exception trace buffers will alter the contents of the **GC History** section in any Javadumps.

Note: When **exception** trace is entered for an active tracepoint, the current thread id is checked against the previous caller's thread id. If it is a different thread, or this is the first call to **exception** trace, a context tracepoint is put into the trace buffer first. This context tracepoint consists only of the current thread id. This is necessary because of the single set of buffers for exception trace. (The formatter identifies all trace entries as coming from the "Exception trace pseudo thread" when it formats **exception** trace files.)

external
The **external** option channels trace data to registered trace listeners in real-time. JVMRI is used to register or deregister as a trace listener. If no listeners are registered, this form of trace does nothing except waste machine cycles on each activated tracepoint.

none

-Xtrace:none prevents the trace engine from loading if it is the only trace option specified. However, if other **-Xtrace** options are on the command line, it is treated as the equivalent of **-Xtrace:none=all** and the trace engine will still be loaded.

If you specify other tracepoints without specifying **-Xtrace:none**, the tracepoints are added to the default set.

Examples

- Default options applied:
  ```
  java -Xrealtime
  ```
- No effect apart from ensuring that the trace engine is loaded (which is the default behavior):
  ```
  java -Xrealtime -Xtrace
  ```
- Trace engine is not loaded:
  ```
  java -Xrealtime -Xtrace:none
  ```
- Trace engine is loaded, but no tracepoints are captured:
  ```
  java -Xrealtime -Xtrace:none=all
  ```
- Default options applied, with the addition of printing for j9vm.209
  ```
  java -Xrealtime -Xtrace:iprint=j9vm.209
  ```
- Default options applied, with the addition of printing for j9vm.209 and j9vm.210. Note the use of brackets when specifying multiple tracepoints.
  ```
  java -Xtrace:iprint={j9vm.209,j9vm.210}
  ```
- Printing for j9vm.209 only:
  ```
  java -Xrealtime -Xtrace:none -Xtrace:iprint=j9vm.209
  ```
- Printing for j9vm.209 only:
  ```
  java -Xrealtime -Xtrace:none,iprint=j9vm.209
  ```
- Default tracing for all components except j9vm, with printing for j9vm.209:
  ```
  java -Xrealtime -Xtrace:none=j9vm,iprint=j9vm.209
  ```
- Default tracing for all components except j9vm, with printing for j9vm.209
  ```
  java -Xrealtime -Xtrace:none=j9vm -Xtrace:iprint=j9vm.209
  ```
- No tracing for j9vm (none overrides iprint):
  ```
  java -Xrealtime -Xtrace:iprint=j9vm.209,none=j9vm
  ```

Tracepoint specification:

You enable tracepoints by specifying *component* and *tracepoint*.

If no qualifier parameters are entered, all tracepoints are enabled, except for **exception.output** trace, where the default is **all {exception}**.

The <*tracepoint_specification*> is as follows:

```
[!]<component>[{<type>}] or [!]<tracepoint_id>[,...]
```

where:

! is a logical not. That is, the tracepoints that are specified immediately following the **!** are turned off.

<*component*>
is one of:
- all
- The JVM subcomponent (that is, dg, j9trc, j9vm, j9mm, j9bcu, j9vrb, j9shr, j9prt, java,awt, awt_dnd_datatransfer, audio, mt, fontmanager, net, awt_java2d, awt_print, or nio)

<*type*> is the tracepoint type or **group**. The following types are supported:
- Entry
- Exit
- Event
- Exception
- Mem
- A group of tracepoints that have been specified by use of a group name. For example, nativeMethods select the group of tracepoints in MT (Method Trace) that relate to native methods. The following groups are supported:
  - compiledMethods
  - nativeMethods
  - staticMethods

<*tracepoint_id*>
is the tracepoint identifier. This constitutes the component name of the tracepoint, followed by its integer number inside that component. For example, j9mm.49, j9shr.20-29, j9vm.15, To understand these numbers, see "Determining the tracepoint ID of a tracepoint" on page 230.

Some tracepoints can be both an exit and an exception; that is, the function ended with an error. If you specify either exit or exception, these tracepoints will be included.

The following tracepoint specification used in Java 5.0 and earlier IBM SDKs is still supported:

```
[!]tpnid{<tracepoint_id>[,...]}
```

Examples

- All tracepoints:
  ```
  -Xtrace:maximal
  ```
- All tracepoints except j9vrb and j9trc:
  ```
  -Xtrace:minimal={all,!j9vrb,!j9trc}
  ```
- All entry and exit tracepoints in j9bcu:
  ```
  -Xtrace:maximal={j9bcu{entry},j9bcu{exit}}
  ```
- All tracepoints in j9mm except tracepoints 20-30:
  ```
  -Xtrace:maximal=j9mm,maximal=!j9mm.20-30
  ```
- Tracepoints j9prt.5 through j9prt.15:
  ```
  -Xtrace:print=j9prt.5-15
  ```

- All j9trc tracepoints:
  ```
  -Xtrace:count=j9trc
  ```
- All entry and exit tracepoints:
  ```
  -Xtrace:external={all{entry},all{exit}}
  ```
- All exception tracepoints:
  ```
  -Xtrace:exception
    or
  -Xtrace:exception=all{exception}
  ```
- All exception tracepoints in j9bcu:
  ```
  -Xtrace:exception=j9bcu
  ```
- Tracepoints j9prt.15 and j9shr.12:
  ```
  -Xtrace:exception={j9prt.15,j9shr.12}
  ```

Trace levels:

Tracepoints have been assigned levels 0 through 9 that are based on the importance of the tracepoint.

A level 0 tracepoint is very important and is reserved for extraordinary events and errors; a level 9 tracepoint is in-depth component detail. To specify a given level of tracing, the level0 through level9 keywords are used. You can abbreviate these keywords to l0 through l9. For example, if level5 is selected, all tracepoints that have levels 0 through 5 are included. Level specifications do not apply to explicit tracepoint specifications that use the TPNID keyword.

The level is provided as a modifier to a component specification, for example:

```
-Xtrace:maximal={all{level5}}
```

or

```
-Xtrace:maximal={j9mm{L2},j9trc,j9bcu{level9},all{level1}}
```

In the first example, tracepoints that have a level of 5 or below are enabled for all components. In the second example, all level 1 tracepoints are enabled, as well as all level2 tracepoints in j9mm, and all tracepoints up to level 9 are enabled in j9bcu. Note that the level applies only to the current component, therefore, if multiple trace selection components are found in a trace properties file, the level is reset to the default for each new component.

Level specifications do not apply to explicit tracepoint specifications that use the TPNID keyword.

When the not operator is specified, the level is inverted; that is, !j9mm{level5} disables all tracepoints of level 6 or above for the j9mm component. For example:

```
-Xtrace:print={all,!j9trc{l5},!j9mm{l6}}
```

enables trace for all components at level 9 (the default), but disables level 6 and above for the locking component, and level 7 and above for the storage component.

Examples

- Count all level zero and one tracepoints hit:
  ```
  -Xtrace:count=all{L1}
  ```
- Produce maximal trace of all components at level 5 and j9mm at level 9:

```
-Xtrace:maximal={all{level5},j9mm{L9}}
```

- Trace all components at level 6, but do not trace j9vrb at all, and do not trace the entry and exit tracepoints in the j9trc component:

```
-Xtrace:minimal={all{l6},!j9vrb,!j9trc{entry},!j9trc{exit}}
```

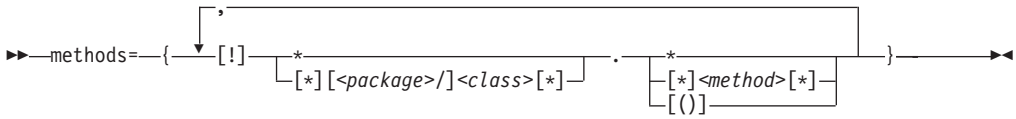
method=<*method_specification*>**[,**<*method_specification*>**]:**

Using method trace provides a complete (and potentially large) diagnosis of code paths inside your application and the system classes. Use wild cards and filtering to control method trace so that you can focus on the sections of code that interest you.

Method trace can trace:

- Method entry
- Method exit

The `methods` parameter is defined as:



Where:

- The delimiter between parts of the package name is a forward slash, "/".
- The **!** in the methods parameter is a NOT operator that allows you to tell the JVM not to trace the specified method or methods.
- The parentheses, **()**, define whether or not to include method parameters in the trace.
- If a method specification includes any commas, the whole specification must be enclosed in braces, for example:

```
-Xtrace:methods={java/lang/*,java/util/*},print=mt
```

- It might be necessary to enclose your command line in quotation marks to prevent the shell intercepting and fragmenting comma-separated command lines, for example:

```
"-Xtrace:methods={java/lang/*,java/util/*},print=mt"
```

To output all method trace information to stderr, use:

-Xtrace:print=mt,methods=*.*
Print method trace information for all methods to stderr.

-Xtrace:iprint=mt,methods=*.*
Print method trace information for all methods to stderr using indentation.

To output method trace information in binary format, see "output=<*filename*>[,*size*m[,<*generations*>]]" on page 223.

Examples

- **Tracing entry and exit of all methods in a given class:**

```
-Xtrace:methods={ReaderMain.*,java/lang/String.*},print=mt
```

This traces all method entry and exit of the ReaderMain class in the default package and the java.lang.String class.

- **Tracing entry, exit and input parameters of all methods in a class:**

  ```
  -Xtrace:methods=ReaderMain.*(),print=mt
  ```

  This traces all method entry, exit, and input of the ReaderMain class in the default package.

- **Tracing all methods in a given package:**

  ```
  -Xtrace:methods=com/ibm/socket/*.*(),print=mt
  ```

  This traces all method entry, exit, and input of all classes in the package com.ibm.socket.

- **Multiple method trace:**

  ```
  -Xtrace:methods={Widget.*(),common/*},print=mt
  ```

  This traces all method entry, exit, and input in the Widget class in the default package and all method entry and exit in the common package.

- **Using the ! operator**

  ```
  -Xtrace:methods={ArticleUI.*,!ArticleUI.get*},print=mt
  ```

  This traces all methods in the ArticleUI class in the default package except those beginning with “get”.

Example output

```
java "-Xtrace:methods={java/lang*.*},iprint=mt" HW
10:02:42.281*0x9e900      mt.4         > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.281 0x9e900      mt.4          > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.281 0x9e900      mt.4           > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.281 0x9e900      mt.4            > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.281 0x9e900      mt.10           < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.281 0x9e900      mt.10          < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.281 0x9e900      mt.4           > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.281 0x9e900      mt.10          < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.281 0x9e900      mt.10         < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.281 0x9e900      mt.4          > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.281 0x9e900      mt.4           > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.296 0x9e900      mt.10          < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.296 0x9e900      mt.10         < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.296 0x9e900      mt.4          > java/lang/String.<clinit>()V Compiled static method
10:02:42.296 0x9e900      mt.4           > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.296 0x9e900      mt.4            > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.296 0x9e900      mt.4              > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.296 0x9e900      mt.10             < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
```

```
10:02:42.296 0x9e900      mt.4             > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.328 0x9e900      mt.10             < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.328 0x9e900      mt.10            < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.328 0x9e900      mt.4             > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.328 0x9e900      mt.10            < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.328 0x9e900      mt.4             > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.328 0x9e900      mt.10            < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.328 0x9e900      mt.10           < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                              V Compiled static method
```

The output lines comprise of:

- `0x9e900`, the current **execenv** (execution environment). Because every JVM thread has its own **execenv**, you can regard **execenv** as a thread-id. All trace with the same **execenv** relates to a single thread.
- The individual tracepoint id in the mt component that collects and emits the data.
- The remaining fields show whether a method is being entered (>) or exited (<), followed by details of the method.

output=<*filename*>**[,***size***m[,**<*generations*>**]]:**

Use the output option to send trace data to <*filename*>. If the file does not already exist, it is created automatically. If it does already exist, it is overwritten.

Optionally:

- You can limit the file to *size* MB, at which point it wraps to the beginning. If you do not limit the file, it grows indefinitely, until limited by disk space.
- If you want the final trace filename to contain today's date, the PID number that produced the trace, or the time, do one of the following steps as appropriate (see also the examples at the end of this section).
  - To include today's date (in "yyyymmdd" format) in the trace filename, specify “%d” as part of the <*filename*>.
  - To include the pidnumber of the process that is generating the tracefile, specify “%p” as part of the <*filename*>.
  - To include the time (in 24-hour hhmmss format) in the trace filename, specify “%t” as part of the <*filename*>.
- You can specify generations as a value 2 through 36. These values cause up to 36 files to be used in a round-robin way when each file reaches its size threshold. When a file needs to be reused, it is overwritten. If generations is specified, the filename must contain a "#" (hash, pound symbol), which will be substituted with its generation identifier, the sequence of which is 0 through 9 followed by A through Z.

Note: When tracing to a file, buffers for each thread are written when the buffer is full or when the JVM terminates. If a thread has been inactive for a period of time before JVM termination, what seems to be 'old' trace data is written to the file. When formatted, it then seems that trace data is missing from the other threads,

but this is an unavoidable side-effect of the buffer-per-thread design. This effect becomes especially noticeable when you use the generation facility, and format individual earlier generations.

Examples

- Trace output goes to /u/traces/gc.problem; no size limit:
  ```
  -Xtrace:output=/u/traces/gc.problem,maximal=j9gc
  ```
- Output goes to trace and will wrap at 2 MB:
  ```
  -Xtrace:output={trace,2m},maximal=j9gc
  ```
- Output goes to gc0.trc, gc1.trc, gc2.trc, each 10 MB in size:
  ```
  -Xtrace:output={gc#.trc,10m,3},maximal=j9gc
  ```
- Output filename contains today's date in yyyymmdd format (for example, traceout.20041025.trc):
  ```
  -Xtrace:output=traceout.%d.trc,maximal=j9gc
  ```
- Output file contains the number of the process (the PID number) that generated it (for example, tracefrompid2112.trc):
  ```
  -Xtrace:output=tracefrompid%p.trc,maximal=j9gc
  ```
- Output filename contains the time in hhmmss format (for example, traceout.080312.trc):
  ```
  -Xtrace:output=traceout.%t.trc,maximal=j9gc
  ```

exception.output=<*filename*>**[,***nnn***m]:**

Use the exception option to redirect exception trace data to <*filename*>.

If the file does not already exist, it is created automatically. If it does already exist, it is overwritten. Optionally, you can limit the file to nnn MB, at which point it wraps nondestructively to the beginning. If you do not limit the file, it grows indefinitely, until limited by disk space.

Optionally, if you want the final trace filename to contain today's date, the PID number that produced the trace, or the time, do one of the following steps as appropriate (see also the examples at the end of this section).

- To include today's date (in “yyyymmdd” format) in the trace filename, specify “%d” as part of the <*filename*>.
- To include the pidnumber of the process that is generating the tracefile, specify “%p”" as part of the <*filename*>.
- To include the time (in 24-hour hhmmss format) in the trace filename, specify “%t” as part of the <*filename*>.

Examples

- Trace output goes to `/u/traces/exception.trc`. No size limit:
  ```
  -Xtrace:exception.output=/u/traces/exception.trc,maximal
  ```
- Output goes to except and wraps at 2 MB:
  ```
  -Xtrace:exception.output={except,2m},maximal
  ```
- Output filename contains today's date in yyyymmdd format (for example, `traceout.20041025.trc`):
  ```
  -Xtrace:exception.output=traceout.%d.trc,maximal
  ```
- Output file contains the number of the process (the PID number) that generated it (for example, `tracefrompid2112.trc`):
  ```
  -Xtrace:exception.output=tracefrompid%p.trc,maximal
  ```

- Output filename contains the time in hhmmss format (for example, `traceout.080312.trc`):

  ```
  -Xtrace:exception.output=traceout.%t.trc,maximal
  ```

resume:

Resumes tracing globally.

Note that suspend and resume are not recursive. That is, two suspends that are followed by a single resume cause trace to be resumed.

Example

- Trace resumed (not much use as a startup option):

  ```
  -Xtrace:resume
  ```

resumecount=<*count*>**:**

This trace option determines whether tracing is enabled for each thread.

If <*count*> is greater than zero, each thread initially has its tracing disabled and must receive <*count*> resume this action before it starts tracing.

Note: You cannot use **resumecount** and **suspendcount** together because they both set the same internal counter.

This system property is for use with the **trigger** property. For more information, see "trigger=<*clause*>[,<*clause*>][,<*clause*>]..." on page 226.

Example

- Start with all tracing turned off. Each thread starts tracing when it has had three **resumethis** actions performed on it:

  ```
  -Xtrace:resumecount=3
  ```

sleeptime=nnn | aaams | bbbs:

Specify how long the sleep lasts when using the sleep trigger action.

Purpose

Use this option to determine how long a sleep trigger action lasts. The default length of time is 30 seconds. If no units are specified, the default time unit is milliseconds.

Parameters

nnn
: Sleep for *nnn* milliseconds.

*aaa***ms**
: Sleep for *aaa* milliseconds.

*bbb***s**
: Sleep for *bbb* seconds.

stackdepth=<n>:

Used to limit the amount of stack frame information collected.

Purpose

Use this option to limit the maximum number of stack frames reported by the jstacktrace trace trigger action. All stack frames are recorded by default.

Parameters

n Record *n* stack frames

suspend:

Suspends tracing globally (for all threads and all forms of tracing) but leaves tracepoints activated.

Example

- Tracing suspended:

```
-Xtrace:suspend
```

suspendcount=<*count*>**:**

This trace option determines whether tracing is enabled for each thread.

If <*count*> is greater than zero, each thread initially has its tracing enabled and must receive <*count*> suspend this action before it stops tracing.

Note: You cannot use **resumecount** and **suspendcount** together because they both set the same internal counter.

This trace option is for use with the **trigger** option. For more information, see "trigger=<*clause*>[,<*clause*>][,<*clause*>]...."

Example

- Start with all tracing turned on. Each thread stops tracing when it has had three **suspendthis** actions performed on it:

```
-Xtrace:suspendcount=3
```

trigger=<*clause*>**[,**<*clause*>**][,**<*clause*>**]...:**

This trace option determines when various triggered trace actions occur. Supported actions include turning tracing on and off for all threads, turning tracing on or off for the current thread, or producing various dumps.

This trace option does not control what is traced. It controls only whether the information that has been selected by the other trace options is produced as normal or is blocked.

Each clause of the **trigger** option can be **tpnid{...}**, **method{...}**, or **group{...}**. You can specify multiple clauses of the same type if required, but you do not have to specify all types. The clause types are as follows:

method{<*methodspec*>[,<*entryAction*>[,<*exitAction*>[,<*delayCount*>[,<*matchcount*>]]]]}
On entering a method that matches <*methodspec*>, the specified <*entryAction*> is run. On leaving it, perform the specified <*exitAction*>. If you specify a <*delayCount*>, the actions are performed only after a matching <*methodspec*> has been entered that many times. If you specify a <*matchCount*>, <*entryAction*> and <*exitAction*> are performed at most that many times.

group{<*groupname*>,<*action*>[,<*delayCount*>[,<*matchcount*>]]}
On finding any active tracepoint that is defined as being in trace group <*groupname*>, for example **Entry** or **Exit**, the specified action is run. If you specify a <*delayCount*>, the action is performed only after that many active tracepoints from group <*groupname*> have been found. If you specify a <*matchCount*>, <*action*> is performed at most that many times.

tpnid{<*tpnid*>|<*tpnidRange*>,<*action*>[,<*delayCount*>[,<*matchcount*>]]}
On finding the specified active <*tpnid*> (tracepoint ID) or a <*tpnid*> that falls inside the specified <*tpnidRange*>, the specified action is run. If you specify a <*delayCount*>, the action is performed only after the JVM finds such an active <*tpnid*> that many times. If you specify a <*matchCount*>, <*action*> is performed at most that many times.

Actions

Wherever an action must be specified, you must select from these choices:

abort
Halt the JVM.

coredump
See **sysdump**.

heapdump
Produce a Heapdump. See "Using Heapdump" on page 187.

javadump
Produce a Javadump. See "Using Javadump" on page 174.

jstacktrace
Walk the Java stack of the current thread and generate auxiliary tracepoints for each stack frame. The auxiliary tracepoints are written to the same destination as the tracepoint or method trace that triggered the action. You can control the number of stack frames walked with the **stackdepth=n** option. See "stackdepth=<n>" on page 225. The **jstacktrace** action is available from Java 6 SR 5.

resume
Resume all tracing (except for threads that are suspended by the action of the **resumecount** property and Trace.suspendThis() calls).

resumethis
Decrement the suspend count for this thread. If the suspend count is zero or below, resume tracing for this thread.

segv
Cause a segmentation violation. (Intended for use in debugging.)

sleep
Delay the current thread for a length of time controlled by the **sleeptime** option. The default is 30 seconds. See "sleeptime=nnn|aaams|bbbs" on page 225.

snap
Snap all active trace buffers to a file in the current working directory. The file name has the format: `Snapnnnn.yyyymmdd.hhmmssth.ppppp.trc`, where `nnnn` is the sequence number of the snap file since JVM startup, `yyyymmdd` is the date, `hhmmssth` is the time, and `ppppp` is the process ID in decimal with leading zeros removed.

suspend
Suspend all tracing (except for special trace points).

suspendthis
Increment the suspend count for this thread. If the suspend-count is greater than zero, prevent all tracing for this thread.

sysdump (or coredump)
Produce a system dump. See "Using system dumps and the dump viewer" on page 192.

Examples

- To start tracing this thread when it enters any method in java/lang/String, and to stop tracing the thread after exiting the method:

```
-Xtrace:resumecount=1
-Xtrace:trigger=method{java/lang/String.*,resumethis,suspendthis}
```

- To resume all tracing when any thread enters a method in any class that starts with "error":

```
-Xtrace:trigger=method{*.error*,resume}
```

- To produce a core dump when you reach the 1000th and 1001st tracepoint from the "jvmri" trace group.

  Note: Without <*matchcount*>, you risk filling your disk with coredump files.

```
-Xtrace:trigger=group{staticmethods,coredump,1000,2}
```

  If using the **trigger** option generates multiple dumps in rapid succession (more than one per second), specify a dump option to guarantee unique dump names. See "Using dump agents" on page 159 for more information.

- To trace (all threads) while the application is active; that is, not start up or shut down. (The application name is "HelloWorld"):

```
-Xtrace:suspend,trigger=method{HelloWorld.main,resume,suspend}
```

- To print a Java stack trace to the console when the `mycomponent.1` tracepoint is reached:

```
-Xtrace:print=mycomponent.1,trigger=tpnid{mycomponent.1,jstacktrace}
```

- To write a Java stack trace to the trace output file when the Sample.code() method is called:

```
-Xtrace:maximal=mt,output=trc.out,methods={mycompany/mypackage/Sample.code},trigger=method{mycompa
```

Using the Java API

You can dynamically control trace in a number of ways from a Java application by using the com.ibm.jvm.Trace class.

Activating and deactivating tracepoints

```
int  set(String cmd);
```

The Trace.set() method allows a Java application to select tracepoints dynamically. For example:

```
Trace.set("iprint=all");
```

The syntax is the same as that used in a trace properties file for the print, iprint, count, maximal, minimal and external trace options.

A single trace command is parsed per invocation of Trace.set, so to achieve the equivalent of **-Xtrace:maximal=j9mm,iprint=j9shr** two calls to Trace.set are needed with the parameters `maximal=j9mm` and `iprint=j9shr`

Obtaining snapshots of trace buffers

```
void snap();
```

You must have activated trace previously with the **maximal** or **minimal** options and without the **out** option.

Suspending or resuming trace

```
void suspend();
```

The `Trace.suspend()` method suspends tracing for all the threads in the JVM.

```
void resume();
```

The `Trace.resume()` method resumes tracing for all threads in the JVM. It is not recursive.

```
void suspendThis();
```

The Trace.suspendThis() method decrements the suspend and resume count for the current thread and suspends tracing the thread if the result is negative.

```
void resumeThis();
```

The Trace.resumeThis() method increments the suspend and resume count for the current thread and resumes tracing the thread if the result is not negative.

Using the trace formatter

The trace formatter is a Java program that converts binary trace point data in a trace file to a readable form. The formatter requires the `J9TraceFormat.dat` file, which contains the formatting templates. The formatter produces a file containing header information about the JVM that produced the binary trace file, a list of threads for which trace points were produced, and the formatted trace points with their timestamp, thread ID, trace point ID and trace point data.

To use the trace formatter on a binary trace file generated in Real Time mode type:

```
java -Xrealtime com.ibm.jvm.format.TraceFormat <input_file> [<output_file>] [options]
```

where <*input_file*> is the name of the binary trace file to be formatted and <*output_file*> is the name of the output file.

To use the trace formatter on a binary trace file generated in standard mode type:

```
java com.ibm.jvm.format.TraceFormat <input_file>  [<output_file>] -datdir <SDK directory>/sdk/jre/
```

where <*input_file*> is the name of the binary trace file to be formatted, <*SDK directory*> is the installation directory of the IBM SDK for Java, and <*output_file*> is the name of the output file.

If you do not specify an output file, the output file is called <*input_file*>`.fmt`.

The size of the heap needed to format the trace is directly proportional to the number of threads present in the trace file. For large numbers of threads the formatter might run out of memory, generating the error `OutOfMemoryError`. In this case, increase the heap size using the **-Xmx** option.

Available options

The following options are available with the trace formatter:

-datdir <*directory*>
Selects an alternative formatting template file directory. The directory must contain the `J9TraceFormat.dat` file.

-help
Displays usage information.

-indent
Indents trace messages at each Entry trace point and outdents trace messages at each Exit trace point. The default is not to indent the messages.

-overridetimezone <*hours*>
Add <*hours*> hours to formatted tracepoints, the value can be negative. This option allows the user to override the default time zone used in the formatter (UTC).

-summary
Prints summary information to the screen without generating an output file.

-threads <*thread id*>**[,**<*thread id*>**]...**
Filters the output for the given thread IDs only. Any number of thread IDs can be specified, separated by commas.

-uservmid <*string*>
Inserts <*string*> in each formatted tracepoint. The string aids reading or parsing when several different JVMs or JVM runs are traced for comparison.

Determining the tracepoint ID of a tracepoint

Throughout the code that makes up the JVM, there are numerous tracepoints. Each tracepoint maps to a unique id consisting of the name of the component containing the tracepoint, followed by a period (".") and then the numeric identifier of the tracepoint.

These tracepoints are also recorded in two `.dat` files (`TraceFormat.dat` and `J9TraceFormat.dat`) that are shipped with the JRE and the trace formatter uses these files to convert compressed trace points into readable form.

JVM developers and Service can use the two `.dat` files to enable formulation of trace point ids and ranges for use under **-Xtrace** when tracking down problems. The next sample taken from the top of `TraceFormat.dat`, which illustrates how this mechanism works:

```
5.1
....
j9bcu.0 0 1 1 N Trc_BCU_VMInitStages_Event1 "  Trace engine initialized for module j9dyn"
j9bcu.1 2 1 1 N Trc_BCU_internalDefineClass_Entry " >internalDefineClass %p"
j9bcu.2 4 1 1 N Trc_BCU_internalDefineClass_Exit " <internalDefineClass %p ->"
j9bcu.3 2 1 1 N Trc_BCU_createRomClassEndian_Entry " >createRomClassEndian searchFilename=%s"
```

The first line of the `.dat` file is an internal version number. Following the version number is a line for each tracepoint. Trace point j9bcu.0 maps to Trc_BCU_VMInitStages_Event1 for example and j9bcu.2 maps to Trc_BCU_internalDefineClass_Exit.

The format of each tracepoint entry is:

```
<component> <t> <o> <l> <e> <symbol> <template>
```

where:

<*component*>
is the SDK component name.

<*t*> is the tracepoint type (0 through 12), where these types are used:
- 0 = event
- 1 = exception

- 2 = function entry
- 4 = function exit
- 5 = function exit with exception
- 8 = internal
- 12 = assert

<*o*> is the overhead (0 through 10), which determines whether the tracepoint is compiled into the runtime JVM code.

<*l*> is the level of the tracepoint (0 through 9). High frequency tracepoints, known as hot tracepoints, are assigned higher level numbers.

<*e*> is an internal flag (Y/N) and no longer used.

<*symbol*>
is the internal symbolic name of the tracepoint.

<*template*>
is a template in double quotation marks that is used to format the entry.

For example, if you discover that a problem occurred somewhere close to the issue of Trc_BCU_VMInitStages_Event, you can rerun the application with **-Xtrace:print=tpnid{j9bcu.0}**. That command will result in an output such as:

```
14:10:42.717*0x41508a00   j9bcu.0        - Trace engine initialized for module j9dyn
```

The example given is fairly trivial. However, the use of tpnid ranges and the formatted parameters contained in most trace entries provides a very powerful problem debugging mechanism.

The `.dat` files contain a list of all the tracepoints ordered by component, then sequentially numbered from 0. The full tracepoint id is included in all formatted output of a tracepoint; For example, tracing to the console or formatted binary trace.

The format of trace entries and the contents of the `.dat` files are subject to change without notice. However, the version number should guarantee a particular format.

Application trace

Application trace allows you to trace Java applications using the JVM Trace Facility.

You must register your Java application with application trace and add trace calls where appropriate. After you have started an application trace module, you can enable or disable individual tracepoints at any time.

Implementing application trace

Application trace is in the package com.ibm.jvm.Trace. The application trace API is described in this section.

Registering for trace:

Use the `registerApplication()` method to specify the application to register with application trace.

The method is of the form:

```
int registerApplication(String application_name, String[] format_template)
```

The `application_name` argument is the name of the application you want to trace. The name must be the same as the application name you specify at JVM startup. The `format_template` argument is an array of format strings like the strings used

by the printf method. You can specify templates of up to 16 KB. The position in the array determines the tracepoint identifier (starting at 0). You can use these identifiers to enable specific tracepoints at runtime. The first character of each template is a digit that identifies the type of tracepoint. The tracepoint type can be one of entry, exit, event, exception, or exception exit. After the tracepoint type character, the template has a blank character, followed by the format string.

The trace types are defined as static values within the Trace class:

```
public static final String EVENT= "0 ";
public static final String EXCEPTION= "1 ";
public static final String ENTRY= "2 ";
public static final String EXIT= "4 ";
public static final String EXCEPTION_EXIT= "5 ";
```

The `registerApplication()` method returns an integer value. Use this value in subsequent `trace()` calls. If the `registerApplication()` method call fails for any reason, the value returned is -1.

Tracepoints:

These trace methods are implemented.

void trace(int handle, int traceId);
void trace(int handle, int traceId, String s1);
void trace(int handle, int traceId, String s1, String s2);
void trace(int handle, int traceId, String s1, String s2, String s3);
void trace(int handle, int traceId, String s1, Object o1);
void trace(int handle, int traceId, Object o1, String s1);
void trace(int handle, int traceId, String s1, int i1);
void trace(int handle, int traceId, int i1, String s1);
void trace(int handle, int traceId, String s1, long l1);
void trace(int handle, int traceId, long l1, String s1);
void trace(int handle, int traceId, String s1, byte b1);
void trace(int handle, int traceId, byte b1, String s1);
void trace(int handle, int traceId, String s1, char c1);
void trace(int handle, int traceId, char c1, String s1);
void trace(int handle, int traceId, String s1, float f1);
void trace(int handle, int traceId, float f1, String s1);
void trace(int handle, int traceId, String s1, double d1);
void trace(int handle, int traceId, double d1, String s1);
void trace(int handle, int traceId, Object o1);
void trace(int handle, int traceId, Object o1, Object o2);
void trace(int handle, int traceId, int i1);
void trace(int handle, int traceId, int i1, int i2);
void trace(int handle, int traceId, int i1, int i2, int i3);
void trace(int handle, int traceId, long l1);
void trace(int handle, int traceId, long l1, long l2);
void trace(int handle, int traceId, long l1, long l2, long i3);
void trace(int handle, int traceId, byte b1);
void trace(int handle, int traceId, byte b1, byte b2);
void trace(int handle, int traceId, byte b1, byte b2, byte b3);
void trace(int handle, int traceId, char c1);
void trace(int handle, int traceId, char c1, char c2);
void trace(int handle, int traceId, char c1, char c2, char c3);
void trace(int handle, int traceId, float f1);
void trace(int handle, int traceId, float f1, float f2);
void trace(int handle, int traceId, float f1, float f2, float f3);

```
void trace(int handle, int traceId, double d1);
void trace(int handle, int traceId, double d1, double d2);
void trace(int handle, int traceId, double d1, double d2, double d3);
void trace(int handle, int traceId, String s1, Object o1, String s2);
void trace(int handle, int traceId, Object o1, String s1, Object o2);
void trace(int handle, int traceId, String s1, int i1, String s2);
void trace(int handle, int traceId, int i1, String s1, int i2);
void trace(int handle, int traceId, String s1, long l1, String s2);
void trace(int handle, int traceId, long l1, String s1, long l2);
void trace(int handle, int traceId, String s1, byte b1, String s2);
void trace(int handle, int traceId, byte b1, String s1, byte b2);
void trace(int handle, int traceId, String s1, char c1, String s2);
void trace(int handle, int traceId, char c1, String s1, char c2);
void trace(int handle, int traceId, String s1, float f1, String s2);
void trace(int handle, int traceId, float f1, String s1, float f2);
void trace(int handle, int traceId, String s1, double d1, String s2);
void trace(int handle, int traceId, double d1, String s1, double d2);
```

The handle argument is the value returned by the registerApplication() method. The traceId argument is the number of the template entry starting at 0.

Printf specifiers:

Application trace supports the ANSI C printf specifiers. You must be careful when you select the specifier; otherwise you might get unpredictable results, including abnormal termination of the JVM.

For 64-bit integers, you must use the ll (lower case LL, meaning long long) modifier. For example: `%lld` or `%lli`.

For pointer-sized integers use the z modifier. For example: `%zx` or `%zd`.

Example HelloWorld with application trace:

This code illustrates a “HelloWorld” application with application trace.

```
import com.ibm.jvm.Trace;
public class HelloWorld
{
   static int handle;
   static String[] templates;
   public static void main ( String[] args )
   {
      templates = new String[ 5 ];
      templates[ 0 ] = Trace.ENTRY          + "Entering %s";
      templates[ 1 ] = Trace.EXIT           + "Exiting %s";
      templates[ 2 ] = Trace.EVENT          + "Event id %d, text = %s";
      templates[ 3 ] = Trace.EXCEPTION      + "Exception: %s";
      templates[ 4 ] = Trace.EXCEPTION_EXIT + "Exception exit from %s";

      // Register a trace application called HelloWorld
      handle = Trace.registerApplication( "HelloWorld", templates );

      // Set any tracepoints requested on command line
      for ( int i = 0; i < args.length; i++ )
      {
         System.err.println( "Trace setting: " + args[ i ] );
         Trace.set( args[ i ] );
      }

      // Trace something....
```

```
      Trace.trace( handle, 2, 1, "Trace initialized" );

      // Call a few methods...
      sayHello( );
      sayGoodbye( );
   }
   private static void sayHello( )
   {
      Trace.trace( handle, 0, "sayHello" );
      System.out.println( "Hello" );
      Trace.trace( handle, 1, "sayHello" );
   }

   private static void sayGoodbye( )
   {
      Trace.trace( handle, 0, "sayGoodbye" );
      System.out.println( "Bye" );
      Trace.trace( handle, 4, "sayGoodbye" );
   }
}
```

Using application trace at runtime

At runtime, you can enable one or more applications for application trace.

For example, in the case of the "HelloWorld" application described above:

```
java -Xrealtime HelloWorld iprint=HelloWorld
```

The HelloWorld example uses the Trace.set() API to pass any arguments to trace, enabling all of the HelloWorld tracepoints to be routed to stderr. Starting the HelloWorld application in this way produces the following output:

```
Trace setting: iprint=HelloWorld
09:50:29.417*0x2a08a00 084002   - Event id 1, text = Trace initialized
09:50:29.417 0x2a08a00 084000   > Entering sayHello
Hello
09:50:29.427 0x2a08a00 084001   < Exiting sayHello
09:50:29.427 0x2a08a00 084000   > Entering sayGoodbye
Bye
09:50:29.437 0x2a08a00 084004 * < Exception exit from sayGoodbye
```

You can obtain a similar result by specifying **iprint** on the command line:

```
java -Xrealtime -Xtrace:iprint=HelloWorld HelloWorld
```

See "Options that control tracepoint activation" on page 216 for more details.

Using method trace

Using method trace provides a complete (and potentially large) diagnosis of code paths inside your application and also inside the system classes. Method trace is a powerful tool that allows you to trace methods in any Java code.

You do not have to add any hooks or calls to existing code. Use wild cards and filtering to control method trace so that you can focus on the sections of code that interest you.

Method trace can trace:

- Method entry
- Method exit

Use method trace to debug and trace application code and the system classes provided with the JVM.

While method trace is powerful, it also has a cost. Application throughput will be significantly affected by method trace, proportionally to the number of methods traced. Additionally, trace output is reasonably large and can grow to consume a significant amount of drive space. For instance, full method trace of a “Hello World” application is over 10 MB.

Running with method trace

Control method trace by using the command-line option **-Xtrace:**<*option*>.

To produce method trace you need to set trace options for the Java classes and methods you want to trace. You also need to route the method trace to the destination you require.

You must set the following two options:

1. Use **-Xtrace:methods** to select which Java classes and methods you want to trace.
2. Use either
   - **-Xtrace:print** to route the trace to stderr.
   - **-Xtrace:maximal** and **-Xtrace:output** to route the trace to a binary compressed file using memory buffers.

Use the **methods** parameter to control what is traced. For example, to trace all methods on the String class, set **-Xtrace:methods=java/lang/String.*,print=mt**.

The **methods** parameter is formally defined as follows:

```
-Xtrace:methods=[[!]<method_spec>[,...]]
```

Where <*method_spec*> is formally defined as:

```
{*|[*]<classname>[*]}.{*|[*]<methodname>[*]}[()]
```

Note:

- The symbol "!" in the methods parameter is a NOT operator. Use this symbol to exclude methods from the trace. Use "this" with other **methods** parameters to set up a trace of the form: “trace methods of this type but not methods of that type”.
- The parentheses, (), that are in the <*method_spec*> define whether to trace method parameters.
- If a method specification includes any commas, the whole specification must be enclosed in braces:

  ```
  -Xtrace:methods={java/lang/*,java/util/*},print=mt
  ```

- On Linux, AIX, z/OS, and i5/OS, you might have to enclose your command line in quotation marks. This action prevents the shell intercepting and fragmenting comma-separated command lines:

  ```
  "-Xtrace:methods={java/lang/*,java/util/*},print=mt"
  ```

Use the **print**, **maximal** and **output** options to route the trace to the required destination, where:

- **print** formats the tracepoint data while the Java application is running and writes the tracepoints to stderr.
- **maximal** saves the tracepoints into memory buffers.
- **output** writes the memory buffers to a file, in a binary compressed format.

To produce method trace that is routed to stderr, use the **print** option, specifying **mt** (method trace). For example: **-Xtrace:methods=java/lang/String.*,print=mt**.

To produce method trace that is written to a binary file from the memory buffers, use the **maximal** and **output** options. For example: **-Xtrace:methods=java/lang/String.*,maximal=mt,output=mytrace.trc**.

If you want your trace output to contain only the tracepoints you specify, use the option **-Xtrace:none** to switch off the default tracepoints. For example: **java -Xtrace:none -Xtrace:methods=java/lang/String.*,maximal=mt,output=mytrace.trc <class>**.

Untraceable methods

Internal Native Library (INL) native methods inside the JVM cannot be traced because they are not implemented using JNI. The list of methods that are not traceable is subject to change without notice between releases.

The INL native methods in the JVM include:

```
java.lang.Class.allocateAndFillArray
java.lang.Class.forNameImpl
java.lang.Class.getClassDepth
java.lang.Class.getClassLoaderImpl
java.lang.Class.getComponentType
java.lang.Class.getConstructorImpl
java.lang.Class.getConstructorsImpl
java.lang.Class.getDeclaredClassesImpl
java.lang.Class.getDeclaredConstructorImpl
java.lang.Class.getDeclaredConstructorsImpl
java.lang.Class.getDeclaredFieldImpl
java.lang.Class.getDeclaredFieldsImpl
java.lang.Class.getDeclaredMethodImpl
java.lang.Class.getDeclaredMethodsImpl
java.lang.Class.getDeclaringClassImpl
java.lang.Class.getEnclosingObject
java.lang.Class.getEnclosingObjectClass
java.lang.Class.getFieldImpl
java.lang.Class.getFieldsImpl
java.lang.Class.getGenericSignature
java.lang.Class.getInterfaceMethodCountImpl
java.lang.Class.getInterfaceMethodsImpl
java.lang.Class.getInterfaces
java.lang.Class.getMethodImpl
java.lang.Class.getModifiersImpl
java.lang.Class.getNameImpl
java.lang.Class.getSimpleNameImpl
java.lang.Class.getStackClass
java.lang.Class.getStackClasses
java.lang.Class.getStaticMethodCountImpl
java.lang.Class.getStaticMethodsImpl
java.lang.Class.getSuperclass
java.lang.Class.getVirtualMethodCountImpl
java.lang.Class.getVirtualMethodsImpl
java.lang.Class.isArray
java.lang.Class.isAssignableFrom
java.lang.Class.isInstance
java.lang.Class.isPrimitive
java.lang.Class.newInstanceImpl
java.lang.ClassLoader.findLoadedClassImpl
java.lang.ClassLoader.getStackClassLoader
java.lang.ClassLoader.loadLibraryWithPath
java.lang.J9VMInternals.getInitStatus
java.lang.J9VMInternals.getInitThread
java.lang.J9VMInternals.initializeImpl
```

```
java.lang.J9VMInternals.sendClassPrepareEvent
java.lang.J9VMInternals.setInitStatusImpl
java.lang.J9VMInternals.setInitThread
java.lang.J9VMInternals.verifyImpl
java.lang.J9VMInternals.getStackTrace
java.lang.Object.clone
java.lang.Object.getClass
java.lang.Object.hashCode
java.lang.Object.notify
java.lang.Object.notifyAll
java.lang.Object.wait
java.lang.ref.Finalizer.runAllFinalizersImpl
java.lang.ref.Finalizer.runFinalizationImpl
java.lang.ref.Reference.getImpl
java.lang.ref.Reference.initReferenceImpl
java.lang.reflect.AccessibleObject.checkAccessibility
java.lang.reflect.AccessibleObject.getAccessibleImpl
java.lang.reflect.AccessibleObject.getExceptionTypesImpl
java.lang.reflect.AccessibleObject.getModifiersImpl
java.lang.reflect.AccessibleObject.getParameterTypesImpl
java.lang.reflect.AccessibleObject.getSignature
java.lang.reflect.AccessibleObject.getStackClass
java.lang.reflect.AccessibleObject.initializeClass
java.lang.reflect.AccessibleObject.invokeImpl
java.lang.reflect.AccessibleObject.setAccessibleImpl
java.lang.reflect.Array.get
java.lang.reflect.Array.getBoolean
java.lang.reflect.Array.getByte
java.lang.reflect.Array.getChar
java.lang.reflect.Array.getDouble
java.lang.reflect.Array.getFloat
java.lang.reflect.Array.getInt
java.lang.reflect.Array.getLength
java.lang.reflect.Array.getLong
java.lang.reflect.Array.getShort
java.lang.reflect.Array.multiNewArrayImpl
java.lang.reflect.Array.newArrayImpl
java.lang.reflect.Array.set
java.lang.reflect.Array.setBoolean
java.lang.reflect.Array.setByte
java.lang.reflect.Array.setChar
java.lang.reflect.Array.setDouble
java.lang.reflect.Array.setFloat
java.lang.reflect.Array.setImpl
java.lang.reflect.Array.setInt
java.lang.reflect.Array.setLong
java.lang.reflect.Array.setShort
java.lang.reflect.Constructor.newInstanceImpl
java.lang.reflect.Field.getBooleanImpl
java.lang.reflect.Field.getByteImpl
java.lang.reflect.Field.getCharImpl
java.lang.reflect.Field.getDoubleImpl
java.lang.reflect.Field.getFloatImpl
java.lang.reflect.Field.getImpl
java.lang.reflect.Field.getIntImpl
java.lang.reflect.Field.getLongImpl
java.lang.reflect.Field.getModifiersImpl
java.lang.reflect.Field.getNameImpl
java.lang.reflect.Field.getShortImpl
java.lang.reflect.Field.getSignature
java.lang.reflect.Field.getTypeImpl
java.lang.reflect.Field.setBooleanImpl
java.lang.reflect.Field.setByteImpl
java.lang.reflect.Field.setCharImpl
java.lang.reflect.Field.setDoubleImpl
java.lang.reflect.Field.setFloatImpl
java.lang.reflect.Field.setImpl
```

```
java.lang.reflect.Field.setIntImpl
java.lang.reflect.Field.setLongImpl
java.lang.reflect.Field.setShortImpl
java.lang.reflect.Method.getNameImpl
java.lang.reflect.Method.getReturnTypeImpl
java.lang.String.intern
java.lang.String.isResettableJVM0
java.lang.System.arraycopy
java.lang.System.currentTimeMillis
java.lang.System.hiresClockImpl
java.lang.System.hiresFrequencyImpl
java.lang.System.identityHashCode
java.lang.System.nanoTime
java.lang.Thread.currentThread
java.lang.Thread.getStackTraceImpl
java.lang.Thread.holdsLock
java.lang.Thread.interrupted
java.lang.Thread.interruptImpl
java.lang.Thread.isInterruptedImpl
java.lang.Thread.resumeImpl
java.lang.Thread.sleep
java.lang.Thread.startImpl
java.lang.Thread.stopImpl
java.lang.Thread.suspendImpl
java.lang.Thread.yield
java.lang.Throwable.fillInStackTrace
java.security.AccessController.getAccessControlContext
java.security.AccessController.getProtectionDomains
java.security.AccessController.getProtectionDomainsImpl
org.apache.harmony.kernel.vm.VM.getStackClassLoader
org.apache.harmony.kernel.vm.VM.internImpl
```

Examples of use

Here are some examples of method trace commands and their results.

- **Tracing entry and exit of all methods in a given class:**

  ```
  -Xtrace:methods=java/lang/String.*,print=mt
  ```

  This example traces entry and exit of all methods in the java.lang.String class. The name of the class must include the full package name, using '/' as a separator. The method name is separated from the class name by a dot '.' In this example, '*' is used to include all methods. Sample output:

  ```
  09:39:05.569 0x1a1100 mt.0 > java/lang/String.length()I Bytecode method, This = 8b27d8
  09:39:05.579 0x1a1100 mt.6 < java/lang/String.length()I Bytecode method
  ```

- **Tracing method input parameters:**

  ```
  -Xtrace:methods=java/lang/Thread.*(),print=mt
  ```

  This example traces all methods in the java.lang.Thread class, with the parentheses '()' indicating that the trace should also include the method call parameters. The output includes an extra line, giving the class and location of the object on which the method was called, and the values of the parameters. In this example the method call is Thread.join(long millis,int nanos), which has two parameters:

  ```
  09:58:12.949 0x4236ce00 mt.0 > java/lang/Thread.join(JI)V Bytecode method, This = 8ffd20
  09:58:12.959 0x4236ce00 mt.18 - Instance method receiver: com/ibm/tools/attach/javaSE/AttachHandle
  arguments: ((long)1000,(int)0)
  ```

- **Tracing multiple methods:**

  ```
  -Xtrace:methods={java/util/HashMap.size,java/lang/String.length},print=mt
  ```

  This example traces the size method on the java.util.HashMap class and the length method on the java.lang.String class. The method specification includes

the two methods separated by a comma, with the entire method specification enclosed in braces '{' and '}'. Sample output:

```
10:28:19.296 0x1a1100 mt.0 > java/lang/String.length()I Bytecode method, This = 8c2548
10:28:19.306 0x1a1100 mt.6 < java/lang/String.length()I Bytecode method
10:28:19.316 0x1a1100 mt.0 > java/util/HashMap.size()I Bytecode method, This = 8dd7e8
10:28:19.326 0x1a1100 mt.6 < java/util/HashMap.size()I Bytecode method
```

- **Using the ! (not) operator to select tracepoints:**

  ```
  -Xtrace:methods={java/util/HashMap.*,!java/util/HashMap.put*},print
  ```

  This example traces all methods in the java.util.HashMap class except those beginning with put. Sample output:

  ```
  10:37:42.225 0x1a1100 mt.0 > java/util/HashMap.createHashedEntry(Ljava/lang/Object;II)Ljava/uti
  e0
  10:37:42.246 0x1a1100 mt.6 < java/util/HashMap.createHashedEntry(Ljava/lang/Object;II)Ljava/uti
  10:37:42.256 0x1a1100 mt.1 > java/util/HashMap.findNonNullKeyEntry(Ljava/lang/Object;II)Ljava/u
  d7e0
  10:37:42.266 0x1a1100 mt.7 < java/util/HashMap.findNonNullKeyEntry(Ljava/lang/Object;II)Ljava/u
  ```

Example of method trace output

An example of method trace output.

Sample output using the command **java -Xtrace:iprint=mt,methods=java/lang/*.* -version**:

```
10:02:42.281*0x9e900     mt.4         > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                             V Compiled static method
10:02:42.281 0x9e900     mt.4          > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                             V Compiled static method
10:02:42.281 0x9e900     mt.4           > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                             V Compiled static method
10:02:42.281 0x9e900     mt.4            > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                             V Compiled static method
10:02:42.281 0x9e900     mt.10           < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                             V Compiled static method
10:02:42.281 0x9e900     mt.10          < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                             V Compiled static method
10:02:42.281 0x9e900     mt.4           > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                             V Compiled static method
10:02:42.281 0x9e900     mt.10          < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                             V Compiled static method
10:02:42.281 0x9e900     mt.10         < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                             V Compiled static method
10:02:42.281 0x9e900     mt.4          > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                             V Compiled static method
10:02:42.281 0x9e900     mt.4           > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                             V Compiled static method
10:02:42.296 0x9e900     mt.10          < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                             V Compiled static method
10:02:42.296 0x9e900     mt.10         < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                             V Compiled static method
10:02:42.296 0x9e900     mt.4          > java/lang/String.<clinit>()V Compiled static method
10:02:42.296 0x9e900     mt.4           > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                             V Compiled static method
10:02:42.296 0x9e900     mt.4            > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                             V Compiled static method
10:02:42.296 0x9e900     mt.4             > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                             V Compiled static method
10:02:42.296 0x9e900     mt.10            < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                             V Compiled static method
10:02:42.296 0x9e900     mt.4             > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                             V Compiled static method
10:02:42.328 0x9e900     mt.10            < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                             V Compiled static method
10:02:42.328 0x9e900     mt.10           < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
```

```
                    V Compiled static method
10:02:42.328 0x9e900      mt.4           > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                    V Compiled static method
10:02:42.328 0x9e900      mt.10          < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                    V Compiled static method
10:02:42.328 0x9e900      mt.4           > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                    V Compiled static method
10:02:42.328 0x9e900      mt.10          < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                    V Compiled static method
10:02:42.328 0x9e900      mt.10         < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                    V Compiled static method
```

The output lines comprise:

- `0x9e900`, the current **execenv** (execution environment). Because every JVM thread has its own **execenv**, you can regard **execenv** as a thread-id. All trace with the same **execenv** relates to a single thread.
- The individual tracepoint id in the mt component that collects and emits the data.
- The remaining fields show whether a method is being entered (>) or exited (<), followed by details of the method.

RTJCL Tracing

The RTJCL tracing provides tracing per component.The broad component categories are AsyncEvents,Scopes,Threads and Timers.The RTJCL code related to these components is extensively instrumented with traces. Each trace has an associated Tracepoint ID.

The tracing mechanism allows you to trace:

1. all the tracepoints
2. only the specified tracepoints in each component on the basis of
   - Trace by type of the tracepoint or
   - Trace by tracing Level or
   - Trace by Tracepoint ID

The tracing can be enabled using the following option in the command line:

```
-Dcom.ibm.realtime.trace.Properties=<component>{(<type>|<level>|<tracepoint ID>|<tracepoint ID-range>
```

- where component is one of (ae,scopes,threads,timers)
- type is one of (entry,exit,event,all)
- level is one of(1,2,3)
- tracepoint ID = ID of the tracepoint.
- tracepoint ID-range = range of tracepoint ids For example,0x02-0x05

For example,if you wanted to trace on the basis of following requirements

1. Tracepoints in AsyncEvent and Scopes
2. Only Entry and Exit points in AsyncEvent component
3. The tracepoints ranging from Tracepoint ID #3 to Tracepoint ID #8 in Scopes component.

Then the command line will look like:

```
-Dcom.ibm.realtime.trace.Properties=ae{entry,exit},scopes{0x03-0x08, 0x15,0x21}
```

The trace ouput however needs to be controlled using the **-Xtrace** options example:

```
-Xtrace:print=ae or -Xtrace:maximal=scopes,output=rtTrace#.trc,20,30m
```

To know all the available trace point IDs,a utility has been provided. The following command line prints all the available trace points:

```
java -Xrealtime com.ibm.realtime.trace.TraceUtil
```

This command helps you to understand more on this utility:

```
java -Xrealtime com.ibm.realtime.trace.TraceUtil help
```

JIT and AOT problem determination

You can use command-line options to help diagnose JIT and AOT compiler problems and to tune performance.

Related information

"JIT compilation and performance" on page 137
The JIT is another area that can affect the performance of your program. When deciding whether or not to use JIT compilation, you must make a balance between faster execution and increased processor usage during compilation. When using the JIT, you should consider the implications to real-time behavior.

Diagnosing a JIT or AOT problem

Occasionally, valid bytecodes might compile into invalid native code, causing the Java program to fail. By determining whether the JIT or AOT compiler is faulty and, if so, *where* it is faulty, you can provide valuable help to the Java service team.

About this task

To determine what methods are compiled when the shared class cache is populated, use the **-Xaot:verbose** option on the admincache command-line. For example:

```
admincache -Xrealtime -Xaot:verbose -populate -aot my.jar -cp <My Class Path>
```

This section describes how you can determine if your problem is compiler-related. This section also suggests some possible workarounds and debugging techniques for solving compiler-related problems.

Disabling the JIT or AOT compiler

If you suspect that a problem is occurring in the JIT or AOT compiler, disable compilation to see if the problem remains. If the problem still occurs, you know that the compiler is not the cause of it.

About this task

The JIT compiler is enabled by default. The AOT compiler is also enabled, but, is not active unless shared classes have been enabled. For efficiency reasons, not all methods in a Java application are compiled. The JVM maintains a call count for each method in the application; every time a method is called and interpreted, the

call count for that method is incremented. When the count reaches the compilation threshold, the method is compiled and executed natively.

The call count mechanism spreads compilation of methods throughout the life of an application, giving higher priority to methods that are used most frequently. Some infrequently used methods might never be compiled at all. As a result, when a Java program fails, the problem might be in the JIT or AOT compiler or it might be elsewhere in the JVM.

The first step in diagnosing the failure is to determine *where* the problem is. To do this, you must first run your Java program in purely interpreted mode (that is, with the JIT and AOT compilers disabled).

Procedure

1. Remove any **-Xjit** and **-Xaot** options (and accompanying parameters) from your command line.
2. Use the **-Xint** command-line option to disable the JIT and AOT compilers. For performance reasons, do not use the **-Xint** option in a production environment.

What to do next

Running the Java program with the compilation disabled leads to one of the following:

- The failure remains. The problem is not in the JIT or AOT compiler. In some cases, the program might start failing in a different manner; nevertheless, the problem is not related to the compiler.
- The failure disappears. The problem is most likely in the JIT or AOT compiler.

  If you are not using shared classes, the JIT compiler is at fault. If you are using shared classes, you must determine which compiler is at fault by running your application with only JIT compilation enabled. Run your application with the **-Xnoaot** option instead of the **-Xint** option. This leads to one of the following:
  - The failure remains. The problem is in the JIT compiler. You can also use the **-Xnojit** instead of the **-Xnoaot** option to ensure that only the JIT compiler is at fault.
  - The failure disappears. The problem is in the AOT compiler.

Selectively disabling the JIT compiler

If the failure of your Java program appears to come from a problem with the JIT compiler, you can try to narrow down the problem further.

About this task

By default, the JIT compiler optimizes methods at various optimization levels; that is, different selections of optimizations are applied to different methods, based on their call counts. Methods that are called more frequently are optimized at higher levels. By changing JIT compiler parameters, you can control the optimization level at which methods are optimized, and determine whether the optimizer is at fault and, if it is, which optimization is problematic.

You specify JIT parameters as a comma-separated list, appended to the **-Xjit** option. The syntax is **-Xjit**:<*param1*>,<*param2*>=<*value*>. For example:

```
java -Xjit:verbose,optLevel=noOpt HelloWorld
```

runs the HelloWorld program, enables verbose output from the JIT, and makes the JIT generate native code without performing any optimizations.

Follow these steps to determine which part of the compiler is causing the failure:

Procedure

1. Set the JIT parameter **count=0** to change the compilation threshold to zero. This causes each Java method to be compiled before it is run. Use **count=0** only when diagnosing problems because significantly more rarely-called methods are compiled, which uses more computing resources for compilation, slowing down your application. With **count=0**, your application should fail immediately when the problem area is reached. In some cases, using **count=1** can reproduce the failure more reliably.
2. Add **disableInlining** to the JIT compiler parameters. **disableInlining** disables the generation of larger and more complex code. More aggressive optimizations are not performed. If the problem no longer occurs, use **-Xjit:disableInlining** as a workaround while the Java service team analyzes and fixes the compiler problem.
3. Decrease the optimization levels by adding the **optLevel** parameter, and re-run the program until the failure no longer occurs or you reach the "noOpt" level. For a JIT compiler problem, start with "scorching" and work down the list. The optimization levels are, in decreasing order:
   a. scorching
   b. veryHot
   c. hot
   d. warm
   e. cold
   f. noOpt

What to do next

If one of these settings causes your failure to disappear, you have a workaround that you can use while the Java service team analyzes and fixes the compiler problem. If removing **disableInlining** from the JIT parameter list does not cause the failure to reappear, do so to improve performance. Follow the instructions in "Locating the failing method" to improve the performance of the workaround.

If the failure still occurs at the "noOpt" optimization level, you must disable the JIT compiler as a workaround.

Locating the failing method

When you have determined the lowest optimization level at which the JIT or AOT compiler must compile methods to trigger the failure, you can find out which part of the Java program, when compiled, causes the failure. You can then instruct the compiler to limit the workaround to a specific method, class, or package, allowing the compiler to compile the rest of the program as usual. For JIT compiler failures, if the failure occurs with **-Xjit:optLevel=noOpt**, you can also instruct the compiler to not compile the method or methods that are causing the failure at all.

Before you begin

If you see error output like this example, you can use it to identify the failing method:

```
Unhandled exception
Type=Segmentation error vmState=0x00000000
Target=2_30_20050520_01866_BHdSMr (Linux 2.4.21-27.0.2.EL)
CPU=s390x (2 logical CPUs) (0x7b6a8000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=4148bf20 Signal_Code=00000001
Handler1=00000100002ADB14 Handler2=00000100002F480C InaccessibleAddress=0000000000000000
gpr0=0000000000000006 gpr1=0000000000000006 gpr2=0000000000000000 gpr3=0000000000000006
gpr4=0000000000000001 gpr5=0000000080056808 gpr6=0000010002BCCA20 gpr7=0000000000000000
......
Compiled_method=java/security/AccessController.toArrayOfProtectionDomains([Ljava/lang/Object;
Ljava/security/AccessControlContext;)[Ljava/security/ProtectionDomain;
```

The important lines are:

vmState=0x00000000
Indicates that the code that failed was not JVM runtime code.

Module= or Module_base_address=
Not in the output (might be blank or zero) because the code was compiled by the JIT, and outside any DLL or library.

Compiled_method=
Indicates the Java method for which the compiled code was produced.

About this task

If your output does not indicate the failing method, follow these steps to identify the failing method:

Procedure

1. Run the Java program with the JIT parameters **verbose** and **vlog=**<*filename*> to the **-Xjit** or **-Xaot** option. With these parameters, the compiler lists compiled methods in a log file named <*filename*>.<*date*>.<*time*>.<*pid*>, also called a *limit file*. A typical limit file contains lines that correspond to compiled methods, like:

   ```
   + (hot) java/lang/Math.max(II)I @ 0x10C11DA4-0x10C11DDD
   ```

   Lines that do not start with the plus sign are ignored by the compiler in the following steps and you can remove them from the file. Methods for which AOT code is loaded from the shared class cache start with `+ (AOT load)`.
2. Run the program again with the JIT or AOT parameter **limitFile=**(<*filename*>,<*m*>,<*n*>), where <*filename*> is the path to the limit file, and <*m*> and <*n*> are line numbers indicating the first and the last methods in the limit file that should be compiled. The compiler compiles only the methods listed on lines <*m*> to <*n*> in the limit file. Methods not listed in the limit file and methods listed on lines outside the range are not compiled and no AOT code in the shared data cache for those methods will be loaded. If the program no longer fails, one or more of the methods that you have removed in the last iteration must have been the cause of the failure.
3. Repeat this process using different values for <*m*> and <*n*>, as many times as necessary, to find the minimum set of methods that must be compiled to trigger the failure. By halving the number of selected lines each time, you can perform a binary search for the failing method. Often, you can reduce the file to a single line.

What to do next

When you have located the failing method, you can disable the JIT or AOT compiler for the failing method only. For example, if the method java/lang/Math.max(II)I causes the program to fail when JIT-compiled with **optLevel=hot**, you can run the program with:

```
-Xjit:{java/lang/Math.max(II)I}(optLevel=warm,count=0)
```

to compile only the failing method at an optimization level of "warm", but compile all other methods as usual.

If a method fails when it is JIT-compiled at "noOpt", you can exclude it from compilation altogether, using the **exclude**={<*method*>} parameter:

```
-Xjit:exclude={java/lang/Math.max(II)I}
```

If a method causes the program to fail when AOT code is loaded from the shared data cache, exclude the method from AOT loading using the **exclude**={<*method*>} parameter:

```
-Xaot:exclude={java/lang/Math.max(II)I}
```

AOT methods are only compiled into the shared class cache during the admincache population step. Preventing AOT loading is the best diagnostic approach for problems with these methods.

Identifying JIT and AOT compilation failures

For JIT compiler failures, analyze the error output to determine if a failure occurs when the JIT compiler attempts to compile a method.

If the JVM crashes, and you can see that the failure has occurred in the JIT library (`libj9jit24.so` or `libj9jit25.so`), the JIT compiler might have failed during an attempt to compile a method.

If you see error output like this example, you can use it to identify the failing method:

```
Unhandled exception
Type=Segmentation error vmState=0x00050000
Target=2_30_20051215_04381_BHdSMr (Linux 2.4.21-32.0.1.EL)
CPU=ppc64 (4 logical CPUs) (0xebf4e000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=00000000 Signal_Code=00000001
Handler1=0000007FE05645B8 Handler2=0000007FE0615C20
R0=E8D4001870C00001 R1=0000007FF49181E0 R2=0000007FE2FBCEE0 R3=0000007FF4E60D70
R4=E8D4001870C00000 R5=0000007FE2E02D30 R6=0000007FF4C0F188 R7=0000007FE2F8C290
......
Module=/home/test/sdk/jre/bin/libj9jit24.so
Module_base_address=0000007FE29A6000
......
Method_being_compiled=com/sun/tools/javac/comp/Attr.visitMethodDef(Lcom/sun/tools/javac/tree/
JCTree$JCMethodDecl;)
```

The important lines are:

vmState=0x00050000
: Indicates that the JIT compiler is compiling code. For a list of vmState code numbers, see the table in Javadump "TITLE, GPINFO, and ENVINFO sections" on page 177

Module=/home/test/sdk/jre/bin/libj9jit24.so
: Indicates that the error occurred in `libj9jit24.so`, the JIT compiler module.

Method_being_compiled=
Indicates the Java method being compiled.

If your output does not indicate the failing method, use the **verbose** option with the following additional settings:

```
-Xjit:verbose={compileStart|compileEnd}
```

These **verbose** settings report when the JIT or AOT compiler starts to compile a method, and when it ends. If the JIT or AOT compiler fails on a particular method (that is, it starts compiling, but crashes before it can end), use the **exclude** parameter on the **-Xjit** or **-Xaot** command-line option to exclude it from JIT or AOT compilation (refer to "Locating the failing method" on page 243). For problems with AOT compilation, destroy your shared class cache before using the **exclude** option. If excluding the method prevents the crash, you have a workaround that you can use while the service team corrects your problem.

Identifying AOT compilation failures in non-real-time-mode

AOT problem determination in non-real-time-mode is very similar to JIT problem determination.

About this task

As with the JIT, first run your application with **-Xnoaot** which ensures that the AOT'ed code is not used when running the application.

If this fixes the problem, rebuild the AOT jar files using the same technique as described in "Locating the failing method" on page 243, providing the **-Xaot** option at AOT build time instead of application runtime.

Identifying AOT compilation failures in real-time mode

AOT problem determination uses the admincache tool to locate the problem.

About this task

In contrast to JIT compilation failures, which occur at application run time, AOT compilation failures occur during the admincache population step.

To find where the problem occurs, run the admincache tool with the **-Xnoaot** option. This ensures that the application does not run with ahead-of-time compiled code.

If using the **-Xnoaot** option fixes the problem, examine the output of the original crash. The output provides information identifying which method is the cause of the problem. Look for a line similar to:

```
Method_being_compiled=myAppClass.main(Ljava/lang/String;)V
```

To avoid the problem, this method must be excluded from ahead-of-time compilation. Do this by adding an option to the admincache command line, similar to the following:

```
-Xaot:exclude={myAppClass.main(Ljava/lang/String;)V}
```

This exclusion prevents AOT compilation of the problem method.

Performance of short-running applications

The IBM JIT compiler is tuned for long-running applications typically used on a server. You can use the **-Xquickstart** command-line option in non-real-time mode to improve the performance of short-running applications, especially for applications in which processing is not concentrated into a small number of methods.

-Xquickstart causes the JIT compiler to use a lower optimization level by default and to compile fewer methods. Performing fewer compilations more quickly can improve application startup time. When the AOT compiler is active (both shared classes and AOT compilation enabled), **-Xquickstart** causes all methods selected for compilation to be AOT compiled, which improves the startup time of subsequent runs. **-Xquickstart** can degrade performance if it is used with long-running applications that contain hot methods. The implementation of **-Xquickstart** is subject to change in future releases.

You can also try improving startup times by adjusting the JIT threshold (using trial and error). See "Selectively disabling the JIT compiler" on page 242 for more information.

-Xquickstart has no effect on AOT code usage with **-Xrealtime**.

JVM behavior during idle periods

You can reduce the CPU cycles consumed by an idle JVM by using the **-XsamplingExpirationTime** option to turn off the JIT sampling thread.

The JIT sampling thread profiles the running Java application to discover commonly used methods. The memory and processor usage of the sampling thread is negligible, and the frequency of profiling is automatically reduced when the JVM is idle.

In some circumstances, you might want no CPU cycles consumed by an idle JVM. To do so, specify the **-XsamplingExpirationTime**<*time*> option. Set <*time*> to the number of seconds for which you want the sampling thread to run. Use this option with care; after it is turned off, you cannot reactivate the sampling thread. Allow the sampling thread to run for long enough to identify important optimizations.

The Diagnostics Collector

The Diagnostics Collector gathers the Java diagnostics files for a problem event.

Using the Diagnostics Collector

The Diagnostics Collector gathers the Java diagnostics files for a problem event.

The Java runtime produces multiple diagnostics files in response to events such as General Protection Faults, out of memory conditions or receiving unexpected operating system signals. The Diagnostics Collector runs just after the Java runtime produces diagnostics files. It searches for system dumps, Java dumps, heap dumps, Java trace dumps and the verbose GC log that match the time stamp for the problem event. If a system dump is found, then optionally the Diagnostics Collector can execute jextract to post-process the dump and capture extra information required to analyze system dumps. The Diagnostics Collector then produces a single .zip file containing all the diagnostics for the problem event.

Steps in the collection of diagnostics are logged in a text file. At the end of the collection process, the log file is copied into the output .zip file.

The Diagnostics Collector also has a feature to give warnings if there are JVM settings in place that could prevent the JVM from producing diagnostics. These warnings are produced at JVM start up so that the JVM can be restarted with fixed settings if necessary. The warnings are printed on `stderr` and in the Diagnostics Collector log file. Fix the settings identified by any warning messages before restarting your Java application. Fixing warnings makes it more likely that the right data is available for IBM Support to diagnose a Java problem.

Using the -Xdiagnosticscollector option

This option enables the Diagnostics Collector.

The Diagnostics Collector is off by default and is enabled by a JVM command-line option:

```
-Xdiagnosticscollector[:settings=<filename>]
```

Specifying a Diagnostics Collector settings file is optional. By default, the settings file `jre/lib/dc.properties` is used. See "Diagnostics Collector settings" on page 250 for details of the settings available.

If you run a Java program from the command line with the Diagnostics Collector enabled, it produces some console output. The Diagnostics Collector runs asynchronously, in a separate process to the one that runs your Java program. The effect is that output appears after the command-line prompt returns from running your program. If this happens, it does not mean that the Diagnostics Collector has hung. Press `enter` to get the command-line prompt back.

Collecting diagnostics from Java runtime problems

The Diagnostics Collector produces an output file for each problem event that occurs in your Java application.

When you add the command-line option **-Xdiagnosticscollector**, the Diagnostics Collector runs and produces several output .zip files. One file is produced at startup. Another file is produced for each dump event that occurs during the lifetime of the JVM. For each problem event that occurs in your Java application, one .zip file is created to hold all the diagnostics for that event. For example, an application might have multiple `OutOfMemoryError`s but keep on running. Diagnostics Collector produces multiple .zip files, each holding the diagnostics from one `OutOfMemoryError`.

The output .zip file is written to the current working directory by default. You can specify a different location by setting the output.dir property in the settings file, as described in "Diagnostics Collector settings" on page 250. An output .zip file name takes the form:

```
java.<event>.<YYYYMMDD.hhmmss.pid>.zip
```

In this file name, *<event>* is one of the following names:

- abortsignal
- check
- dumpevent
- gpf

- outofmemoryerror
- usersignal
- vmstart
- vmstop

These event names refer to the event that triggered Diagnostics Collector. The name provides a hint about the type of problem that occurred. The default name is *dumpevent*, and is used when a more specific name cannot be given for any reason.

<*YYYYMMDD.hhmmss.pid*> is a combination of the time stamp of the dump event, and the process ID for the original Java application. *pid* is not the process ID for the Diagnostics Collector.

The Diagnostics Collector copies files that it writes to the output .zip file. It does not delete the original diagnostics information.

When the Diagnostics Collector finds a system dump for the problem event, then by default it runs jextract to post-process the dump and gather context information This information enables later debugging. Diagnostics Collector automates a manual step that is requested by IBM support on most platforms. You can prevent Diagnostics Collector from running jextract by setting the property **run.jextract** to **false** in the settings file. For more information, see "Diagnostics Collector settings" on page 250.

The Diagnostics Collector logs its actions and messages in a file named `JavaDiagnosticsCollector.<number>.log`. The log file is written to the current working directory. The log file is also stored in the output .zip file. The <*number*> component in the log file name is not significant; it is added to keep the log file names unique.

The Diagnostics Collector is a Java VM dump agent. It is run by the Java VM in response to the dump events that produce diagnostic files by default. It runs in a new Java process, using the same version of Java as the VM producing dumps. This ensures that the tool runs the correct version of jextract for any system dumps produced by the original Java process.

Verifying your Java diagnostics configuration

When you enable the command-line option -Xdiagnosticscollector, a diagnostics configuration check runs at Java VM start up. If any settings disable key Java diagnostics, a warning is reported.

The aim of the diagnostics configuration check is to avoid the situation where a problem occurs after a long time, but diagnostics are missing because they were inadvertently switched off. Diagnostic configuration check warnings are reported on `stderr` and in the Diagnostics Collector log file. A copy of the log file is stored in the `java.check.<timestamp>.<pid>.zip` output file.

If you do not see any warning messages, it means that the Diagnostics Collector has not found any settings that disable diagnostics. The Diagnostics Collector log file stored in `java.check.<timestamp>.<pid>.zip` gives the full record of settings that have been checked.

For extra thorough checking, the Diagnostics Collector can trigger a Java dump. The dump provides information about the command-line options and current Java system properties. It is worth running this check occasionally, as there are

command-line options and Java system properties that can disable significant parts of the Java diagnostics. To enable the use of a Java dump for diagnostics configuration checking, set the **config.check.javacore** option to **true** in the settings file. For more information, see "Diagnostics Collector settings."

For all platforms, the diagnostics configuration check examines environment variables that can disable Java diagnostics. For reference purposes, the full list of current environment variables and their values is stored in the Diagnostics Collector log file.

Checks for operating system settings are carried out on Linux and AIX. On Linux, the core and file size ulimits are checked. On AIX, the settings **fullcore=true** and **pre430core=false** are checked, as well as the core and file size ulimits.

Configuring the Diagnostics Collector

The Diagnostics Collector supports various options that can be set in a properties file.

Diagnostics Collector can be configured by using options that are set in a properties file. By default, the properties file is `jre/lib/dc.properties`. If you do not have access to edit this file, or if you are working on a shared system, you can specify an alternative filename using:

```
-Xdiagnosticscollector:settings=<filename>
```

Using a settings file is optional. By default, Diagnostics Collector gathers all the main types of Java diagnostics files.

Diagnostics Collector settings

The Diagnostics Collector has several settings that affect the way the collector works.

The settings file uses the standard Java properties format. It is a text file with one **property=value** pair on each line. Each supported property controls the Diagnostic Collector in some way. Lines that start with '#' are comments.

Parameters

file.<any_string>=<pathname>

Any property with a name starting **file.** specifies the path to a diagnostics file to collect. You can add any string as a suffix to the property name, as a reminder of which file the property refers to. You can use any number of **file.** properties, so you can tell the Diagnostics Collector to collect a list of custom diagnostic files for your environment. Using **file.** properties does not alter or prevent the collection of all the standard diagnostic files. Collection of standard diagnostic files always takes place.

Custom debugging scripts or software can be used to produce extra output files to help diagnose a problem. In this situation, the settings file is used to identify the extra debug output files for the Diagnostics Collector. The Diagnostics Collector collects the extra debug files at the point when a problem occurs. Using the Diagnostics Collector in this way means that debug files are collected immediately after the problem event, increasing the chance of capturing relevant context information.

output.dir=<output_directory_path>

The Diagnostic Collector tries to write its output .zip file to the output directory path that you specify. The path can be absolute or relative to the working directory of the Java process. If the directory does not exist, the Diagnostics Collector tries to create it. If the directory cannot be created, or the directory is not writeable, the Diagnostics Collector defaults to writing its output .zip file to the current working directory.

Note: On Windows systems, Java properties files use backslash as an escape character. To specify a backslash as part of Windows path name, use a double backslash '\\' in the properties file.

loglevel.file=<level>
This setting controls the amount of information written to the Diagnostic Collector log file. The default setting for this property is **config**. Valid levels are:

off No information reported.

severe Errors are reported.

warning
Report warnings in addition to information reported by **severe**.

info More detailed information in addition to that reported by **warning**.

config Configuration information reported in addition to that reported by **info**. This is the default reporting level.

fine Tracing information reported in addition to that reported by **config**.

finer Detailed tracing information reported in addition to that reported by **fine**.

finest Report even more tracing information in addition to that reported by **finer**.

all Report everything.

loglevel.console=<level>
Controls the amount of information written by the Diagnostic Collector to `stderr`. Valid values for this property are as described for loglevel.file. The default setting for this property is **warning**.

settings.id=<identifier>
Allows you to set an identifier for the settings file. If you set loglevel.file to **fine** or **lower**, the **settings.id** is recorded in the Diagnostics Collector log file as a way to check that your settings file is loaded as expected.

config.check.javacore={true | false}
Set **config.check.javacore=true** to enable a Java dump for the diagnostics configuration check at virtual machine start-up. The check means that the virtual machine start-up takes more time, but it enables the most thorough level of diagnostics configuration checking.

run.jextract=false
Set this option to prevent the Diagnostics Collector running jextract on detected System dumps.

Known limitations

There are some known limitations for the Diagnostics Collector.

If Java programs do not start at all on your system, for example because of a Java runtime installation problem or similar issue, the Diagnostics Collector cannot run.

The Diagnostics Collector does not respond to additional **-Xdump** settings that specify extra dump events requiring diagnostic information. For example, if you use **-Xdump** to produce dumps in response to a particular exception being thrown, the Diagnostics Collector does not collect the dumps from this event.

Garbage Collector diagnostics

This section describes how to diagnose garbage collection.

For information about Real Time garbage collection diagnostics, see "Troubleshooting the Metronome Garbage Collector" on page 51. For information about garbage collection diagnostics in the standard JVM, see the *Diagnostics Guide*.

Shared classes diagnostics

Understanding how to diagnose problems that might occur will help you to use shared classes mode.

For an introduction to shared classes, see Class data sharing between JVMs.

The topics that are discussed in this chapter are:

Some of the material in this section might not be applicable to real-time mode shared class caches. In real-time mode, applications only have readonly access to shared class caches. Caches can be modified exclusively using the admincache tool. Nonpersistent caches are not available in real-time mode.

Deploying shared classes

You cannot just "switch on" class sharing without considering how to deploy it sensibly for the chosen application. This section looks at some of the important issues to consider.

Cache naming

If multiple users will be using an application that is sharing classes or multiple applications are sharing the same cache, knowing how to name caches appropriately is important. The ultimate goal is to have the smallest number of caches possible, while maintaining secure access to the class data and allowing as many applications and users as possible to share the same classes.

To use a cache for a specific application, write the cache into the application installation directory using the **-Xshareclasses:cachedir=**<*dir*> option. This helps prevent users of other applications from accidentally using the same cache, and automatically removes the cache if the application is uninstalled.

If the same user will always be using the same application, either use the default cache name (which includes the user name) or specify a cache name specific to the application. The user name can be incorporated into a cache name using the %u modifier, which causes each user running the application to get a separate cache.

On Linux, AIX, z/OS, and i5/OS platforms, if multiple users in the same operating system group are running the same application, use the **groupAccess** suboption, which creates the cache allowing all users in the same primary group to share the same cache. If multiple operating system groups are running the same application, the %g modifier can be added to the cache name, causing each group running the application to get a separate cache.

Multiple applications or different JVM installations can share the same cache provided that the JVM installations are of the same service release level. It is possible for different JVM service releases to share the same cache, but it is not advised. The JVM will attempt to destroy and re-create a cache created by a different service release. See "Compatibility between service releases" on page 258 for more information.

Small applications that load small numbers of application classes should all try to share the same cache, because they will still be able to share bootstrap classes. For large applications that contain completely different classes, it might be more sensible for them to have a class cache each, because there will be few common classes and it is then easier to selectively clean up caches that aren't being used.

On Linux, AIX, z/OS, and i5/OS, `/tmp` is used as the default directory, which is shared by all users.

Cache access

A JVM can access a shared class cache with either read-write or read-only access. Read-write access is the default and allows all users equal rights to update the cache. Use the **-Xshareclasses:readonly** option for read-only access.

Opening a cache as read-only makes it easier to administrate operating system permissions. A cache created by one user cannot be opened read-write by other users, but other users can get startup time benefits by opening the cache as read-only. Opening a cache as read-only also prevents corruption of the cache. This can be useful on production systems where one instance of an application corrupting the cache could affect the performance of all other instances.

When a cache is opened read-only, class files of the application that are modified or moved cannot be updated in the cache. Sharing will be disabled for the modified or moved containers for that JVM.

Related information

"Security considerations for the shared class cache" on page 7
The shared class cache is designed for ease of cache management and usability, but the default security policy might not be appropriate.

Cache housekeeping

Unused caches on a system use resources that could be used by another application. Ensuring that caches are sensibly managed is important.

The JVM offers a number of features to assist in cache housekeeping. To understand these features, it is important to explain the differences in behavior between persistent and non-persistent caches.

Persistent caches are written to disk and remain there until explicitly destroyed. Persistent caches are not removed when the operating system is restarted. Because persistent caches do not exist in shared memory, the only penalty of not destroying stale caches is that they take up disk space.

Non-persistent caches exist in shared memory. They retain system resources that could usefully be employed by other applications. However, non-persistent caches are automatically purged when the operating system is restarted, so housekeeping is only an issue between operating system restarts.

The success of any housekeeping functions is dependent on the user having the correct operating system permissions, whether the housekeeping is automatic or explicit. In general, if the user has the permissions to open a cache with read-write access, they also have the permissions to destroy it. The only exception is for non-persistent caches on Linux, AIX, z/OS, and i5/OS. These caches can only be destroyed by the user which created the cache. Caches can only be destroyed if they are not in use.

The JVM provides a number of housekeeping utilities, which are all suboptions to the **-Xshareclasses** command-line option. Each suboption performs the explicit action requested. The suboption might also perform other automated housekeeping activities. Each suboption works in the context of a specific **cacheDir**.

destroy
This suboption destroys a named cache

destroyAll
This suboption destroys all caches in the specified **cacheDir**.

expire=<*time in minutes*>
This suboption looks for caches which have not been connected to for the <*time in minutes*> specified. If any caches are found which have not been connected to in that specified time, they are destroyed.

expire=0
This suboption is the same as **destroyAll**.

expire=10000
This suboption destroys all caches which have not been used for approximately one week.

There is also a certain amount of automatic housekeeping which is done by the JVM. Most of this automatic housekeeping is driven by the cache utilities. **destroyAll** and **expire** attempt to destroy all persistent and non-persistent caches of all JVM levels and service releases in a given **cacheDir**. **destroy** only works on a specific cache of a specific name and type.

There are two specific cases where the JVM attempts automatic housekeeping when not requested by the user.

1. The first case is when a JVM connects to a cache, and determines that the cache is corrupted or was created by a different service release. The JVM attempts to destroy and re-create the cache.
2. The second case is if `/tmp/javasharedresources` is deleted on a Linux, AIX, z/OS, or i5/OS system. The JVM attempts to identify leaked shared memory areas from non-persistent caches. If any areas are found, they are purged.

With persistent caches, it is safe to delete the cache files manually from the file system. Each persistent cache has only one system object: the cache file.

It is not safe to delete cache files manually for non-persistent caches. The reason is that each non-persistent cache has four system objects: a shared memory area, a shared semaphore, and two control files to identify the memory and semaphores to the JVM. Deleting the control files causes the memory and semaphores to be leaked. They can then only be identified and removed using the ipcs and ipcrm commands on Linux, AIX, z/OS, and i5/OS.

The **reset** suboption can also be used to cause a JVM to refresh an existing class cache when it starts up. The cache is destroyed and re-created if it is not already in use. The option **-Xshareclasses:reset** can be added anywhere to the command line. The option does not override any other **Xshareclasses** command-line options. This constraint means that **-Xshareclasses:reset** can be added to the **IBM_JAVA_OPTIONS** environment variable, or any of the other means of passing command-line options to the JVM.

Cache performance

Shared classes employs numerous optimizations to perform as well as possible under most circumstances. However, there are configurable factors which can affect shared classes performance, which are discussed here.

Use of Java archive and compressed files

The cache keeps itself up-to-date with file system updates by constantly checking file system timestamps against the values in the cache.

Because a classloader can obtain a lock on a `.jar` file, after the `.jar` has been opened and read, it is assumed that the `.jar` remains locked and does not need to be constantly checked.

Because `.class` files can created or deleted from a directory at any time, a directory in a class path, particularly near the start, will inevitably have a performance affect on shared classes because it must be constantly checked for classes that might have created. For example, with a class path of `/dir1:jar1.jar:jar2.jar:jar3.jar;`, when loading any class from the cache using this class path, the directory `/dir1` must be checked for the existence of the class for every class load. This checking also requires fabricating the expected directory from the class's package name. This operation can be expensive.

Advantages of not filling the cache

A full shared classes cache is not a problem for any JVMs connected to it. However, a full cache can place restrictions on how much sharing can be performed by other JVMs or applications.

ROMClasses are added to the cache and are all unique. Metadata is added describing the ROMClasses and there can be multiple metadata entries corresponding to a single ROMClass. For example, if class A is loaded from myApp1.jar and then another JVM loads the same class A from myOtherApp2.jar, only one ROMClass will exist in the cache, with two pieces of metadata describing the two locations it came from.

If many classes are loaded by an application and the cache is 90% full, another installation of the same application can use the same cache, and the amount of extra information that needs to be added about the second application's classes is minimal, even though they are separate copies on the file system.

After the extra metadata has been added, both installations can share the same classes from the same cache. However, if the first installation fills the cache completely, there is no room for the extra metadata and the second installation cannot share classes because it cannot update the cache. The same limitation applies for classes that become stale and are redeemed. (See "Redeeming stale classes" on page 264). Redeeming the stale class requires a small quantity of metadata to be added to the cache. If you cannot add to the cache, because it is full, the class cannot be redeemed.

Read-only cache access

If the JVM opens a cache with read-only access, it does not need to obtain any operating system locks to read the data, which can make cache access slightly faster. However, if any containers of cached classes are changed or moved on a class path, then sharing will be disabled for all classes on that class path. This is because the JVM is unable to update the cache with the changes and it is too expensive for the cache code to continually re-check for updates to containers on each class-load.

Page protection

By default, the JVM protects all cache memory pages using page protection to prevent accidental corruption by other native code running in the process. If any native code attempts to write to the protected page, the process will exit, but all other JVMs will be unaffected.

The only page not protected by default is the cache header page because the cache header must be updated much more frequently than the other pages. The cache header can be protected by using the **-Xshareclasses:mprotect=all** option. This has a very small affect on performance and is not enabled by default.

Switching off memory protection completely using **-Xshareclasses:mprotect=none** does not provide significant performance gains.

Caching Ahead Of Time (AOT) code

The JVM might automatically store a small amount of Ahead Of Time (AOT) compiled native code in the cache when it is populated with classes. The AOT code allows any subsequent JVMs attaching to the cache to start faster. AOT data is generated for methods where it is likely to be most effective.

You can use the **-Xshareclasses:noaot**, **-Xscminaot**, and **-Xscmaxaot** options to control the use of AOT code in the cache.

In general, the default settings provide significant startup performance benefits and use only a small amount of cache space. In some cases, for example, running the JVM without the JIT, there is no benefit gained from the cached AOT code. In these cases you should turn off caching of AOT code.

To diagnose AOT issues, use the **-Xshareclasses:verboseAOT** command-line option. This will generate messages when AOT code is found or stored in the cache, and extra messages you can use to detect cache problems related to AOT. These messages all begin with the code JVMJITM.

Making the most efficient use of cache space

A shared class cache is a finite size and cannot grow. The JVM attempts to make the most efficient use of cache space that it can. It does this by sharing strings between classes and ensuring that classes are not duplicated. However, there are also command-line options which allow the user to optimize the cache space available.

-Xscminaot and **-Xscmaxaot** place upper and lower limits on the amount of AOT data the JVM can store in the cache and **-Xshareclasses:noaot** prevents the JVM from storing any AOT data.

-Xshareclasses:nobootclasspath disables the sharing of classes on the boot class path, so that only classes from application classloaders are shared. There are also optional filters that can be applied to Java classloaders to place custom limits on the classes that are added to the cache.

Very long class paths

When a class is loaded from the shared class cache, the class path against which it was stored and the class path of the caller classloader are "matched" to see whether the cache should return the class. The match does not have to be exact, but the result should be exactly the same as if the class were loaded from disk.

Matching very long class paths is initially expensive, but successful and failed matches are remembered, so that loading classes from the cache using very long class paths is much faster than loading from disk.

Growing classpaths

Where possible, avoid gradually growing a classpath in a URLClassLoader using addURL(). Each time an entry is added, an entire new class path must be added to the cache.

For example, if a class path with 50 entries is grown using addURL(), you could create 50 unique class paths in the cache. This gradual growth uses more cache space and has the potential to slow down class path matching when loading classes.

Concurrent access

A shared class cache can be updated and read concurrently by any number of JVMs.

Any number of JVMs can read from the cache at the same time as a single JVM is writing to it. If many JVMs start at the same time and no cache exists, one JVM will win the race to create the cache and then all JVMs will race to populate the cache with potentially the same classes.

Multiple JVMs concurrently loading the same classes are coordinated to a certain extent by the cache itself to mitigate the effects of many JVMs loading the same class from disk and racing to store it.

Class GC with shared classes

Running with shared classes has no affect on class garbage collection. Classloaders loading classes from the shared class cache can be garbage collected in exactly the same way as classloaders that load classes from disk. If a classloader is garbage collected, the ROMClasses it has added to the cache will persist.

Compatibility between service releases

Use the most recent service release of a JVM for any application.

It is not recommended for different service releases to share the same class cache concurrently. A class cache is compatible with earlier and later service releases. However, there might be small changes in the class files or the internal class file format between service releases. These changes might result in duplication of classes in the cache. For example, a cache created by a given service release can continue to be used by an updated service release, but the updated service release might add extra classes to the cache if space allows.

To reduce class duplication, if the JVM connects to a cache which was created by a different service release, it attempts to destroy the cache then re-create it. This automated housekeeping feature is designed so that when a new JVM level is used with an existing application, the cache is automatically refreshed. However, the refresh only succeeds if the cache is not in use by any other JVM. If the cache is in use, the JVM cannot refresh the cache, but uses it where possible.

If different service releases do use the same cache, the JVM disables AOT. The effect is that AOT code in the cache is ignored.

Nonpersistent shared cache cleanup

When using UNIX System V workstations, you might want to clean up the cache files manually.

When using nonpersistent caches on UNIX System V workstations, four artifacts are created on the system:

- Some System V shared memory.
- A System V semaphore.
- A control file for the shared memory.
- A control file for the semaphore.

The control files are used to look up the System V IPC objects. For example, the semaphore control file provides information to help find the System V semaphore. During system cleanup, ensure that you do not delete the control files before the System V IPC objects are removed.

To remove artifacts, run a J9 JVM with the **-Xsharedclasses:nonpersistent,destroy** or **-Xsharedclasses:destroyAll** command-line options. For example:

```
java –Xshareclasses:nonpersistent,destroy,name=mycache
```

or

```
java –Xshareclasses:destroyAll
```

It is sometimes necessary to clean up a system manually, for example when the control files have been removed from the file system.

For Java 6 SR4 and later, manual cleanup is required when the JVM warns that you are attaching to a System V object that might be orphaned because of a missing control file. For example, you might see messages like the following output:

```
JVMPORT021W You have opened a stale System V shared semaphore: file:/tmp/javasharedresources/C240D
JVMPORT020W You have opened a stale System V shared memory: file:/tmp/javasharedresources/C240D2A6
```

J9 JVMs earlier than Java 6 SR4 produce error messages like the following to indicate a problem with the system:

```
JVMSHRC020E An error has occurred while opening semaphore
JVMSHRC017E Error code: -308
JVMSHRC320E Error recovery: destroying shared memory semaphores.
JVMJ9VM015W Initialization error for library j9shr24(11):
JVMJ9VM009E J9VMDllMain failed
```

In response to these messages, run the following command as root, or for each user that might have created shared caches on the system:

```
ipcs -a
```

- For Java 6 SR4 and later, record all semaphores IDs with corresponding keys having MSB 0xad.
- For Java 6 SR4 and later, record all memory IDs with corresponding keys having MSB 0xde.
- For earlier versions of Java 6, do the same except both keys begin with MSB 0x01 to 0x14

For each System V semaphore ID, run the command:

```
ipcrm –s <semid>
```

where `<semid>` is the recorded System V semaphore ID.

For each System V shared memory ID, run the command:

```
ipcrm –m <shmid>
```

where `<shmid>` is the recorded System V shared memory ID.

Dealing with runtime bytecode modification

Modifying bytecode at runtime is an increasingly popular way to engineer required function into classes. Sharing modified bytecode improves startup time, especially when the modification being used is expensive. You can safely cache modified bytecode and share it between JVMs, but there are many potential problems because of the added complexity. It is important to understand the features described in this section to avoid any potential problems.

This section contains a brief summary of the tools that can help you to share modified bytecode.

Potential problems with runtime bytecode modification

The sharing of modified bytecode can cause potential problems.

When a class is stored in the cache, the location from which it was loaded and a time stamp indicating version information are also stored. When retrieving a class from the cache, the location from which it was loaded and the time stamp of that location are used to determine whether the class should be returned. The cache does not note whether the bytes being stored were modified before they were

defined unless it is specifically told so. Do not underestimate the potential problems that this modification could introduce:

- In theory, unless all JVMs sharing the same classes are using exactly the same bytecode modification, JVMs could load incorrect bytecode from the cache. For example, if JVM1 populates a cache with modified classes and JVM2 is not using a bytecode modification agent, but is sharing classes with the same cache, it could incorrectly load the modified classes. Likewise, if two JVMs start at the same time using different modification agents, a mix of classes could be stored and both JVMs will either throw an error or demonstrate undefined behavior.
- An important prerequisite for caching modified classes is that the modifications performed must be deterministic and final. In other words, an agent which performs a particular modification under one set of circumstances and a different modification under another set of circumstances, cannot use class caching. This is because only one version of the modified class can be cached for any given agent and once it is cached, it cannot be modified further or returned to its unmodified state.

In practice, modified bytecode can be shared safely if the following criteria are met:

- Modifications made are deterministic and final (described above).
- The cache knows that the classes being stored are modified in a particular way and can partition them accordingly.

The VM provides features that allow you to share modified bytecode safely, for example using "modification contexts". However, if a JVMTI agent is unintentionally being used with shared classes without a modification context, this usage does not cause unexpected problems. In this situation, if the VM detects the presence of a JVMTI agent that has registered to modify class bytes, it forces all bytecode to be loaded from disk and this bytecode is then modified by the agent. The potentially modified bytecode is passed to the cache and the bytes are compared with known classes of the same name. If a matching class is found, it is reused; otherwise, the potentially modified class is stored in such a way that other JVMs cannot load it accidentally. This method of storing provides a "safety net" that ensures that the correct bytecode is always loaded by the JVM running the agent, but any other JVMs sharing the cache will be unaffected. Performance during class loading could be affected because of the amount of checking involved, and because bytecode must always be loaded from disk. Therefore, if modified bytecode is being intentionally shared, the use of modification contexts is recommended.

Modification contexts

A modification context creates a private area in the cache for a given context, so that multiple copies or versions of the same class from the same location can be stored using different modification contexts. You choose the name for a context, but it must be consistent with other JVMs using the same modifications.

For example, one JVM uses a JVMTI agent "agent1", a second JVM uses no bytecode modification, a third JVM also uses "agent1", and a fourth JVM uses a different agent, "agent2". If the JVMs are started using the following command lines (assuming that the modifications are predictable as described above), they should all be able to share the same cache:

```
java -agentlib:agent1 -Xshareclasses:name=cache1,modified=myAgent1 myApp.ClassName
java -Xshareclasses:name=cache1 myApp.ClassName
java -agentlib:agent1 -Xshareclasses:name=cache1,modified=myAgent1 myApp.ClassName
java -agentlib:agent2 -Xshareclasses:name=cache1,modified=myAgent2 myApp.ClassName
```

SharedClassHelper partitions

Modification contexts cause all classes loaded by a particular JVM to be stored in a separate cache area. If you need a more granular approach, the SharedClassHelper API can store individual classes under "partitions".

This ability to use partitions allows an application class loader to have complete control over the versioning of different classes and is particularly useful for storing bytecode woven by Aspects. A partition is a string key used to identify a set of classes. For example, a system might weave a number of classes using a particular Aspect path and another system might weave those classes using a different Aspect path. If a unique partition name is computed for the different Aspect paths, the classes can be stored and retrieved under those partition names.

The default application class loader or bootstrap class loader does not support the use of partitions; instead, a SharedClassHelper must be used with a custom class loader.

Using the safemode option

If you have unexpected results or VerifyErrors from cached classes, use safemode to determine if the bytecode from the cache is correct for your JVM.

Unexpected results from cached classes, or VerifyErrors, might be caused by the wrong classes being returned. Another cause might be incorrect cached classes. You can use a debugging mode called safemode to find whether the bytecode being loaded from the cache is correct for the JVM you are using.

Note: In Java 6, using `-Xshareclasses:safemode` is the same as running `-Xshareclasses:none`. This option has the same effect as not enabling shared classes.

safemode is a suboption of **-Xshareclasses**. It prevents the use of shared classes. safemode does not add classes to a cache.

When you use safemode with a populated cache, it forces the JVM to load all classes from disk and then apply any modifications to those classes. The class loader then tries to store the loaded classes in the cache. The class being stored is compared byte-for-byte against the class that would be returned if the class loader had not loaded the class from disk. If any bytes do not match, the mismatch is reported to `stderr`. Using safemode helps ensure that all classes are loaded from disk. safemode provides a useful way of verifying whether the bytes loaded from the shared class cache are the expected bytes.

Do not use safemode in production systems, because it is only a debugging tool and does not share classes.

JVMTI redefinition and retransformation of classes

Redefined classes are never stored in the cache. Retransformed classes are not stored in the cache by default, but caching can be enabled using the **-Xshareclasses:cacheRetransformed** option.

Redefined classes are classes containing replacement bytecode provided by a JVMTI agent at runtime, typically where classes are modified during a debugging session. Redefined classes are never stored in the cache.

Retransformed classes are classes with registered retransformation capable agents that have been called by a JVMTI agent at runtime. Unlike RedefineClasses, the

RetransformClasses function allows the class definition to be changed without reference to the original bytecode. An example of retransformation is a profiling agent that adds or removes profiling calls with each retransformation. Retransformed classes are not stored in the cache by default, but caching can be enabled using the **-Xshareclasses:cacheRetransformed** option. This option will also work with modification contexts or partitions.

Further considerations for runtime bytecode modification

There are a number of additional items that you need to be aware of when using the cache with runtime bytecode modification.

If bytecode is modified by a non-JVMTI agent and defined using the JVM's application classloader when shared classes are enabled, these modified classes are stored in the cache and nothing is stored to indicate that these are modified classes. Another JVM using the same cache will therefore load the classes with these modifications. If you are aware that your JVM is storing modified classes in the cache using a non-JVMTI agent, you are advised to use a modification context with that JVM to protect other JVMs from the modifications.

Combining partitions and modification contexts is possible but not recommended, because you will have "partitions inside partitions". In other words, a partition A stored under modification context X will be different from partition A stored under modification context B.

Because the shared class cache is a fixed size, storing many different versions of the same class might require a much larger cache than the size that is typically required. However, note that the identical classes are never duplicated in the cache, even across modification contexts or partitions. Any number of metadata entries might describe the class and where it came from, but they all point to the same class bytes.

If an update is made to the file system and the cache marks a number of classes as stale as a result, note that it will mark all versions of each class as stale (when versions are stored under different modification contexts or partitions) regardless of the modification context being used by the JVM that caused the classes to be marked stale.

Understanding dynamic updates

The shared class cache must respond to file system updates; otherwise, a JVM might load from the cache classes that are out of date (stale). After a class has been marked stale, it is not returned by the cache if it is requested by a class loader. Instead, the class loader must reload the class from disk and store the updated version in the cache.

The cache manages itself to ensure that it deals with the following challenges:

- Java archive and compressed files are typically locked by class loaders when they are in use, but can be updated when the JVM shuts down. Because the cache persists beyond the lifetime of any JVM using it, subsequent JVMs connecting to the cache will check for Java archive and compressed file updates.
- .class files (not in jar) can be updated at any time during the lifetime of a JVM. The cache checks for individual class file updates.
- .class files can be created or removed in directories in classpaths at any time during the lifetime of a JVM. The cache checks the classpath for classes that have been created or removed.

- .class files must be in a directory structure that reflects their package structure; therefore, when checking for updates, the correct directories must be searched.

Because class files contained in jars and compressed files and class files stored as .class files on the file system present different challenges, the cache treats these as two different types. Updates are managed by writing file system time stamps into the cache.

Classes found or stored using a SharedClassTokenHelper cannot be maintained in this way, because Tokens are meaningless to the cache. AOT data will be updated automatically as a direct consequence of the class data being updated.

Storing classes

When a classpath is stored in the cache, the Java archive and compressed files are time stamped and these time stamps are stored as part of the classpath. (Directories are not time stamped.) When a ROMClass is stored, if it came from a .class file on the file system, the .class file it came from is time stamped and this time stamp is stored. Directories are not time stamped because there is no guarantee that updates to a file will cause an update to its directory.

If a compressed or Java archive file does not exist, the classpath containing it can still be added to the cache, but ROMClasses from this entry are not stored. If a ROMClass is being added to the cache from a directory and it does not exist as a .class file, it is not stored.

Time stamps can also be used to determine whether a ROMClass being added is a duplicate of one that already exists in the cache.

If a classpath entry is updated on the file system and this entry is out of sync with a classpath time stamp in the cache, the classpath is added again and time stamped again in its entirety. Therefore, when a ROMClass is being added to the cache and the cache is searched for the caller's classpath, any potential classpath matches are also time stamp-checked to ensure that they are up-to-date before the classpath is returned.

Finding classes

When the JVM finds a class in the cache, it has to make more checks than when it stores a class.

When a potential match has been found, if it is a .class file on the file system, the time stamps of the .class file and the ROMClass stored in the cache are compared. Regardless of the source of the ROMClass (jar or .class file), every Java archive and compressed file entry in the caller's classpath, up to and including the index at which the ROMClass was “found”, must be checked for updates by obtaining the time stamps. Any update could mean that another version of the class being returned might have been added earlier in the classpath.

Additionally, any classpath entries that are directories might contain .class files that will “shadow” the potential match that has been found. Class files might be created or deleted in these directories at any point. Therefore, when the classpath is walked and jars and compressed files are checked, directory entries are also checked to see whether any .class files have been created unexpectedly. This check involves building a string out of the classpath entry, the package names, and the class name, and then looking for the classfile. This procedure is expensive if many

directories are being used in class paths. Therefore, using jar files gives better shared classes performance.

Marking classes as stale

When a Java archive or compressed file classpath entry is updated, all of the classes in the cache that could potentially have been affected by that update are marked “stale”. When an individual .class file is updated, only the class or classes stored from that .class file are marked stale.

The stale marking used is pessimistic because the cache does not know the contents of individual jars and compressed files.

For example, therefore, for the following class paths where c has become stale:

a;b;c;d c could now contain new versions of classes in d; therefore, classes in both c and d are all stale.

c;d;a c could now contain new versions of classes in d and a; therefore, classes in c, d, and a are all stale.

Classes in the cache that have been loaded from c, d, and a are marked stale. Therefore, it takes only a single update to one jar file to potentially cause many classes in the cache to be marked stale. To ensure that there is not massive duplication as classes are unnecessarily restored, stale classes can be "redeemed" if it is proved that they are not in fact stale.

Redeeming stale classes

Because classes are marked stale when a class path update occurs, many of the classes marked stale might have not updated. When a class loader stores a class that effectively “updates” a stale class, you can “redeem” the stale class if you can prove that it has not in fact changed.

For example, class X is stored from c with classpath a;b;c;d. Suppose that a is updated, meaning that a could now contain a new version of X (although it does not) but all classes loaded from b, c, and d are marked stale. Another JVM wants to load X, so it asks the cache for it, but it is stale, so the cache does not return the class. The class loader therefore loads it from disk and stores it, again using classpath a;b;c;d. The cache checks the loaded version of X against the stale version of X and, if it matches, the stale version is "redeemed".

AOT code

A single piece of AOT code is associated with a specific method in a specific version of a class in the cache. If new classes are added to the cache as a result of a file system update, new AOT code can be generated for those classes. If a particular class becomes stale, the AOT code associated with that class also becomes stale. If a class is redeemed, the AOT code associated with that class is also redeemed. AOT code is not shared between multiple versions of the same class.

The total amount of AOT code can be limited using **-Xscmaxaot** and cache space can be reserved for AOT code using **-Xscminaot**.

Using the Java Helper API

Classes are shared by the bootstrap class loader internally in the JVM, but any other Java class loader must use the Java Helper API to find and store classes in the shared class cache.

The Helper API provides a set of flexible Java interfaces that enable Java class loaders to exploit the shared classes features in the JVM. The java.net.URLClassLoader shipped with the SDK has been modified to use a SharedClassURLClasspathHelper and any class loaders that extend java.net.URLClassLoader inherit this behavior. Custom class loaders that do not extend URLClassLoader but want to share classes must use the Java Helper API. This section contains a summary on the different types of Helper API available and how to use them.

The Helper API classes are contained in the com.ibm.oti.shared package and Javadoc information for these classes is shipped with the SDK (some of which is reproduced here).

com.ibm.oti.shared.Shared

The Shared class contains static utility methods: getSharedClassHelperFactory() and isSharingEnabled(). If **-Xshareclasses** is specified on the command line and sharing has been successfully initialized, isSharingEnabled() returns true. If sharing is enabled, getSharedClassHelperFactory() will return a com.ibm.oti.shared.SharedClassHelperFactory. The helper factories are singleton factories that manage the Helper APIs. To use the Helper APIs, you must get a Factory.

com.ibm.oti.shared.SharedClassHelperFactory

SharedClassHelperFactory provides an interface used to create various types of SharedClassHelper for class loaders. Class loaders and SharedClassHelpers have a one-to-one relationship. Any attempts to get a helper for a class loader that already has a different type of helper causes a HelperAlreadyDefinedException.

Because class loaders and SharedClassHelpers have a one-to-one relationship, calling findHelperForClassLoader() returns a Helper for a given class loader if one exists.

com.ibm.oti.shared.SharedClassHelper

There are three different types of SharedClassHelper:

- SharedClassTokenHelper. Use this Helper to store and find classes using a String token generated by the class loader. Typically used by class loaders that require complete control over cache contents.
- SharedClassURLHelper. Store and find classes using a file system location represented as a URL. For use by class loaders that do not have the concept of a classpath, that load classes from multiple locations.
- SharedClassURLClasspathHelper. Store and find classes using a classpath of URLs. For use by class loaders that load classes using a URL class path

Compatibility between Helpers is as follows: Classes stored by SharedClassURLHelper can be found using a SharedClassURLClasspathHelper and the opposite also applies. However, classes stored using a SharedClassTokenHelper can be found only by using a SharedClassTokenHelper.

Note also that classes stored using the URL Helpers are updated dynamically by the cache (see “Understanding dynamic updates” on page 262) but classes stored by the SharedClassTokenHelper are not updated by the cache because the Tokens are meaningless Strings, so it has no way of obtaining version information.

You can control the classes a URL Helper will find and store in the cache using a SharedClassURLFilter. An object implementing this interface can be passed to the SharedClassURLHelper when it is constructed or after it has been created. The filter is then used to decide which classes to find and store in the cache. See “SharedClassHelper API” for more information. For a detailed description of each helper and how to use it, see the Javadoc information shipped with the SDK.

com.ibm.oti.shared.SharedClassStatistics
The SharedClassStatistics class provides static utilities that return the total cache size and the amount of free bytes in the cache.

SharedClassHelper API

The SharedClassHelper API provides functions to find and store shared classes.

These functions are:

findSharedClass
Called after the class loader has asked its parent for a class, but before it has looked on disk for the class. If findSharedClass returns a class (as a byte[]), pass this class to defineClass(), which defines the class for that JVM and return it as a java.lang.Class object. The byte[] returned by findSharedClass is not the actual class bytes. The effect is that you cannot instrument or manipulate the bytes in the same way as class bytes loaded from a disk. If a class is not returned by findSharedClass, the class is loaded from disk (as in the nonshared case) and then the java.lang.Class defined is passed to storeSharedClass.

storeSharedClass
Called if the class loader has loaded class bytes from disk and has defined them using defineClass. Do not use storeSharedClass to try to store classes that were defined from bytes returned by findSharedClass.

setSharingFilter
Register a filter with the SharedClassHelper that is to be used to decide which classes are found and stored in the cache. Only one filter can be registered with each SharedClassHelper.

You must resolve how to deal with metadata that cannot be stored. An example is when java.security.CodeSource or java.util.jar.Manifest objects are derived from jar files. For each jar, the recommended way to deal with metadata that cannot be stored is always to load the first class from the jar. Load the class regardless of whether it exists in the cache or not. This load activity initializes the required metadata in the class loader, which can then be cached internally. When a class is then returned by findSharedClass, the function indicates where the class has been loaded from. The result is that the correct cached metadata for that class can be used.

It is not incorrect usage to use storeSharedClass to store classes that were loaded from disk, but which are already in the cache. The cache sees that the class is a duplicate of an existing class, it is not duplicated, and so the class continues to be

shared. However, although it is handled correctly, a class loader that uses only storeSharedClass is less efficient than one that also makes appropriate use of findSharedClass.

Filtering

You can filter which classes are found and stored in the cache by registering an object implementing the SharedClassFilter interface with the SharedClassHelper. Before accessing the cache, the SharedClassHelper functions performs filtering using the registered SharedClassFilter object. For example, you can cache classes inside a particular package only by creating a suitable filter. To define a filter, implement the SharedClassFilter interface, which defines the following methods:

```
boolean acceptStore(String className)
boolean acceptFind(String className)
```

To allow a class to be found or stored in the cache, return true from your implementation of these functions. Your implementation of these functions can use the supplied parameters as required. Make sure that you implement short-running functions because they are called for every find and store. Register a filter on a SharedClassHelper using the setSharingFilter(SharedClassFilter filter) function. See the Javadoc for the SharedClassFilter interface for more information.

Applying a global filter

You can apply a SharedClassFilter to all non-bootstrap class loaders which share classes. Specify the com.ibm.oti.shared.SharedClassGlobalFilterClass system property on the command line. For example:

```
-Dcom.ibm.oti.shared.SharedClassGlobalFilterClass=<filter class name>
```

Understanding shared classes diagnostics output

When running in shared classes mode, a number of diagnostics tools can help you. The verbose options are used at runtime to show cache activity and you can use the printStats and printAllStats utilities to analyze the contents of a shared class cache.

This section tells you how to interpret the output.

Verbose output

The **verbose** suboption of **-Xshareclasses** gives the most concise and simple diagnostic output on cache usage.

Verbose output will typically look like this:

```
 >java -Xshareclasses:name=myCache,verbose -Xscmx10k HelloWorld
[-Xshareclasses verbose output enabled]
JVMSHRC158I Successfully created shared class cache "myCache"
JVMSHRC166I Attached to cache "myCache", size=10200 bytes
JVMSHRC096I WARNING: Shared Cache "myCache" is full. Use -Xscmx to set cache size.
Hello
JVMSHRC168I Total shared class bytes read=0. Total bytes stored=9284
```

This output shows that a new cache called myCache was created, which was only 10 kilobytes in size and the cache filled up almost immediately. The message displayed on shut down shows how many bytes were read or stored in the cache.

VerboseIO output

The verboseIO output is far more detailed and is used at runtime to show classes being stored and found in the cache. You enable verboseIO output by using the **verboseIO** suboption of **-Xshareclasses**.

VerboseIO output provides information about the I/O activity occurring with the cache, with basic information on find and store calls. With a cold cache, you see trace like this:

```
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 0... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 3... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 17... Failed.
Storing class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 17... Succeeded.
```

Each classloader is given a unique ID and the boostrap loader is always 0. In the trace above, you see classloader 17 obeying the classloader hierarchy of asking its parents for the class. Each of its parents consequently asks the shared cache for the class because it does not yet exist in the cache, all the find calls fail and classloader 17 stores it.

After the class is stored, you see the following output:

```
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 0... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 3... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 17... Succeeded.
```

Again, the classloader obeys the hierarchy, with its parents asking the cache for the class first. It succeeds for the correct classloader. Note that with alternative classloading frameworks, such as OSGi, the parent delegation rules are different, so you will not necessarily see this type of output.

VerboseHelper output

You can also obtain diagnostics from the Java SharedClassHelper API using the **verboseHelper** suboption.

The output is divided into information messages and error messages:

- Information messages are prefixed with:

  ```
  Info for SharedClassHelper id <n>: <message>
  ```

- Error messages are prefixed with:

  ```
  Error for SharedClassHelper id <n>: <message>
  ```

Use the Java Helper API to obtain this output; see “Using the Java Helper API” on page 265.

verboseAOT output

VerboseAOT provides output when compiled AOT code is being found or stored in the cache.

When a cache is being populated, you might see the following:

```
Storing AOT code for ROMMethod 0x523B95C0 in shared cache... Succeeded.
```

When a populated cache is being accessed, you might see the following:

```
Finding AOT code for ROMMethod 0x524EAEB8 in shared cache... Succeeded.
```

AOT code is generated heuristically. You might not see any AOT code generated at all for a small application.

printStats utility

The printStats utility is a suboption of **-Xshareclasses**, optionally taking a cache name using **name=**<*name*>. It prints summary information on the cache specified to standard error. Because it is a cache utility, the JVM displays the information on the cache specified and then exits.

Here is a detailed description of what the output means:

```
baseAddress      = 0x53133000
endAddress       = 0x590E0000
allocPtr         = 0x548B2F88

cache size       = 100662924
free bytes       = 63032784
ROMClass bytes   = 32320692
AOT bytes        = 4277036
Data bytes       = 339667
Metadata bytes   = 692745
Metadata % used  = 1%

# ROMClasses = 9576
# AOT Methods = 3136
# Classpaths = 5
# URLs = 111
# Tokens = 0
# Stale classes = 0
% Stale classes = 0%

Cache is 37% full
```

baseAddress and endAddress
: Give the boundary addresses of the shared memory area containing the classes.

allocPtr
: Is the address where ROMClass data is currently being allocated in the cache.

cache size and free bytes
: `cache size` shows the total size of the shared memory area in bytes, and `free bytes` shows the free bytes remaining.

ROMClass bytes
: Is the number of bytes of class data in the cache.

AOT bytes
: Is the number of bytes of Ahead Of Time (AOT) compiled code in the cache.

Data bytes
: Is the number of bytes of non-class data stored by the JVM.

Metadata bytes
: Is the number of bytes of data stored to describe the cached classes.

Metadata % used
: Shows the proportion of metadata bytes to class bytes; this proportion indicates how efficiently cache space is being used.

 # ROMClasses
: Indicates the number of classes in the cache. The cache stores `ROMClasses` (the class data itself, which is read-only) and it also stores information about the location from which the classes were loaded. This information is

stored in different ways, depending on the Java SharedClassHelper API (see “Using the Java Helper API” on page 265) used to store the classes

 # AOT methods
ROMClass methods can optionally be compiled and the AOT code stored in the cache. This information shows the total number of methods in the cache that have AOT code compiled for them. This number includes AOT code for stale classes.

 # Classpaths, URLs, and Tokens
Indicates the number of classpaths, URLs, and tokens in the cache. Classes stored from a SharedClassURLClasspathHelper are stored with a Classpath; those stored using a SharedClassURLHelper are stored with a URL; and those stored using a SharedClassTokenHelper are stored with a Token. Most classloaders (including the bootstrap and application classloaders) use a SharedClassURLClasspathHelper, so it is most common to see Classpaths in the cache. The number of Classpaths, URLs, and Tokens stored is determined by a number of factors. For example, every time an element of a Classpath is updated (for example, a jar is rebuilt), a new Classpath is added to the cache. Additionally, if "partitions" or "modification contexts" are used, these are associated with the Classpath, URL, and Token, and one is stored for each unique combination of partition and modification context.

 # Stale classes
Are classes that have been marked as "potentially stale" by the cache code, because of an operating system update. See “Understanding dynamic updates” on page 262.

% Stale classes
Is an indication of the proportion of classes in the cache that have become stale.

printAllStats utility

The printAllStats utility is a suboption of **-Xshareclasses**, optionally taking a cache name using **name=**<*name*>. This utility lists the cache contents in order. It aims to give as much diagnostic information as possible and, because the output is listed in chronological order, you can interpret it as an "audit trail" of cache updates. Because it is a cache utility, the JVM displays the information on the cache specified or the default cache and then exits.

Each JVM that connects to the cache receives a unique ID and each entry in the output is preceded by a number indicating the JVM that wrote the data.

Classpaths

```
1: 0x2234FA6C CLASSPATH
        C:\myJVM\sdk\jre\lib\vm.jar
        C:\myJVM\sdk\jre\lib\core.jar
        C:\myJVM\sdk\jre\lib\charsets.jar
        C:\myJVM\sdk\jre\lib\graphics.jar
        C:\myJVM\sdk\jre\lib\security.jar
        C:\myJVM\sdk\jre\lib\ibmpkcs.jar
        C:\myJVM\sdk\jre\lib\ibmorb.jar
        C:\myJVM\sdk\jre\lib\ibmcfw.jar
        C:\myJVM\sdk\jre\lib\ibmorbapi.jar
        C:\myJVM\sdk\jre\lib\ibmjcefw.jar
        C:\myJVM\sdk\jre\lib\ibmjgssprovider.jar
        C:\myJVM\sdk\jre\lib\ibmjsseprovider2.jar
        C:\myJVM\sdk\jre\lib\ibmjaaslm.jar
```

```
C:\myJVM\sdk\jre\lib\ibmjaasactivelm.jar
C:\myJVM\sdk\jre\lib\ibmcertpathprovider.jar
C:\myJVM\sdk\jre\lib\server.jar
C:\myJVM\sdk\jre\lib\xml.jar
```

This output indicates that JVM 1 caused a class path to be stored at address `0x2234FA6C` in the cache and that it contains 17 entries, which are listed. If the class path was stored using a given partition or modification context, this information is also displayed.

ROMClasses

```
1: 0x2234F7DC ROMCLASS: java/lang/Runnable at 0x213684A8
        Index 1 in class path 0x2234FA6C
```

This output indicates that JVM 1 stored a class called java/lang/Runnable in the cache. The metadata about the class is stored at address `0x2234F7DC` and the class itself is written to address `0x213684A8`. It also indicates the class path against which the class is stored and from which index in that class path the class was loaded; in this case, the class path is the same address as the one listed above. If a class is stale, it has `!STALE!` appended to the entry. If the ROMClass was stored using a given partition or modification context, this information is also displayed.

AOT methods

```
1: 0x540FBA6A AOT: loadConvert
        for ROMClass java/util/Properties at 0x52345174
```

This output indicates that JVM 1 stored AOT compiled code for the method loadConvert() in java/util/Properties. The ROMClass address is the address of the ROMClass that contains the method that was compiled. If an AOT method is stale, it has `!STALE!` appended to the entry.

URLs and Tokens

These are displayed in the same format as Classpaths. A URL is effectively the same as a Classpath, but with only one entry. A Token is in a similar format, but it is a meaningless String passed to the Java Helper API.

Debugging problems with shared classes

The following sections describe some of the situations you might encounter with shared classes and also the tools that are available to assist in diagnosing problems.

Using shared classes trace

Use shared classes trace output only for debugging internal problems or for a very detailed trace of activity in the shared classes code.

You enable shared classes trace using the **j9shr** trace component as a suboption of **-Xtrace**. See “Tracing Java applications and the JVM” on page 208 for details. Five levels of trace are provided, level 1 giving essential initialization and runtime information, up to level 5, which is very detailed.

Shared classes trace output does not include trace from the port layer functions that deal with memory-mapped files, shared memory and shared semaphores. It also does not include trace from the Helper API natives. Port layer trace is enabled using the **j9prt** trace component and trace for the Helper API natives is enabled using the **j9jcl** trace component.

Why classes in the cache might not be found or stored

This quick guide helps you to diagnose why classes might not be being found or stored in the cache as expected.

Why classes might not be found

The class is stale

As explained in "Understanding dynamic updates" on page 262, if a class has been marked as "stale", it is not returned by the cache.

A JVMTI agent is being used without a modification context

If a JVMTI agent is being used without a modification context, classes cannot be found in the cache. The effect is to give the JVMTI agent an opportunity to modify the bytecode when the classes are loaded from disk. For more information, see "Dealing with runtime bytecode modification" on page 259.

The Classpath entry being used is not yet confirmed by the SharedClassURLClasspathHelper

Class path entries in the SharedClassURLClasspathHelper must be "confirmed" before classes can be found for these entries. A class path entry is confirmed by having a class stored for that entry. For more information about confirmed entries, see the SharedClassHelper Javadoc information.

Why classes might not be stored

The cache is full

The cache is a finite size, determined when it is created. When it is full, it cannot be expanded. When the **verbose** suboption is enabled a message is printed when the cache reaches full capacity, to warn the user. The **printStats** utility also displays the occupancy level of the cache, and can be used to query the status of the cache.

The cache is opened read-only

When the **readonly** suboption is specified, no data is added to the cache.

The class does not exist on the file system

The class might have been generated or might come from a URL location that is not a file.

The class loader does not extend java.net.URLClassLoader

For a class loader to share classes, it must either extend java.net.URLClassLoader or implement the Java Helper API (see "SharedClassHelper API" on page 266)

The class has been retransformed by JVMTI and cacheRetransformed has not been specified

As described in "Dealing with runtime bytecode modification" on page 259, the option **cacheRetransformed** must be selected for retransformed classes to be cached.

The class was generated by reflection or Hot Code Replace

These types of classes are never stored in the cache.

Why classes might not be found or stored

Safemode is being used

Classes are not found or stored in the cache in safemode. This behavior is expected for shared classes. See "Using the safemode option" on page 261.

The cache is corrupted

In the unlikely event that the cache is corrupted, no classes can be found or stored.

A SecurityManager is being used and the permissions have not been granted to the class loader

SharedClassPermissions need to be granted to application class loaders so that they can share classes when a SecurityManager is used. For more information, see the *SDK and Runtime User Guide* for your platform.

Dealing with initialization problems

Shared classes initialization requires a number of operations to succeed. A failure could have many potential reasons and it is difficult to provide detailed information on the command line following an initialization failure. Some common reasons for failure are listed here.

If you cannot see why initialization has failed from the command-line output, look at level 1 trace for more information regarding the cause of the failure. The *SDK and Runtime User Guide* for your platform provides detailed information about operating system limitations, thus only a brief summary of potential reasons for failure is provided here.

Writing data into the `javasharedresources` directory

To initialize any cache, data must be written into a `javasharedresources` directory, which is created by the first JVM that needs it.

On Linux, AIX, z/OS, and i5/OS this directory is `/tmp/javasharedresources`. On Windows it is `C:\Documents and Settings\<username>\Local Settings\Application Data\javasharedresources`.

On Windows, the memory-mapped file is written here. On Linux, AIX, z/OS, and i5/OS this directory is used only to store small amounts of metadata that identify the semaphore and shared memory areas.

Problems writing to this directory are the most likely cause of initialization failure. The default cache name is created with the username incorporated to prevent clashes if different users try to share the same default cache, but all shared classes users must have permissions to write to `javasharedresources`. The user running the first JVM to share classes on a system must have permission to create the `javasharedresources` directory.

By default on Linux, AIX, z/OS, and i5/OS caches are created with user-only access, meaning that two users cannot share the same cache unless the **-Xshareclasses:groupAccess** command-line option is used when the cache is created. If user A creates a cache using **-Xshareclasses:name=myCache** and user B also tries to run the same command line, a failure will occur, because user B does not have permissions to access the existing cache called “myCache”. Caches can be destroyed only by the user who created them, even if **-Xshareclasses:groupAccess** is used.

Initializing a persistent cache

Persistent caches are the default on all platforms except for AIX and z/OS.

The following operations must succeed to initialize a persistent cache:

1) Creating the cache file

Persistent caches are a regular file created on disc. The main reasons for failing to create the file are insufficient disc space and incorrect file permissions.

2) Acquiring file locks

Concurrent access to persistent caches is controlled using operating system file-locking. The main reason for failing to obtain the necessary file locks is attempting to use a cache that is located on a remote networked file system (such as an NFS or SMB mount). This is not supported.

3) Memory-mapping the file

The cache file is memory-mapped so that reading and writing to and from it is a fast operation. The main reasons for failing to memory-map the file are insufficient system memory or attempting to use a cache which is located on a remote networked file system (such as an NFS or SMB mount). This is not supported.

Initializing a non-persistent cache

Non-persistent caches are the default on AIX and z/OS.

The following operations must succeed to initialize a non-persistent cache:

1) Create a shared memory area

Possible problems depend on your platform.

Linux, AIX, z/OS, and i5/OS

The **SHMMAX** operating system environment variable by default is set quite low. **SHMMAX** limits the size of shared memory segment that can be allocated. If a cache size greater than **SHMMAX** is requested, the JVM attempts to allocate **SHMMAX** and outputs a message indicating that **SHMMAX** should be increased. For this reason, the default cache size is 16 MB.

2) Create a shared semaphore

Shared semaphores are created in the `javasharedresources` directory. You must have write access to this directory.

3) Write metadata

Metadata is written to the `javasharedresources` directory. You must have write access to this directory.

If you are experiencing considerable initialization problems, try a hard reset:

1. Run `java -Xshareclasses:destroyAll` to remove all known memory areas and semaphores. On a Linux, AIX, or z/OS system, run this command as root, or as a user with *ALLOBJ authority on i5/OS.
2. Delete the `javasharedresources` directory and all of its contents.
3. On Linux, AIX, z/OS, or i5/OS the memory areas and semaphores created by the JVM might not have been removed using **-Xshareclasses:destroyAll**. This problem is addressed the next time you start the JVM. If the JVM starts and the `javasharedresources` directory does not exist, an automated cleanup is triggered and any remaining shared memory areas that are shared class caches are destroyed. Run the JVM with **-Xshareclasses** as root on Linux, AIX, or z/OS or as a user with *ALLOBJ authority on i5/OS, to ensure that the system is completely reset. The JVM then automatically recreates the `javasharedresources` directory.

Dealing with verification problems

Verification problems (typically seen as java.lang.VerifyErrors) are potentially caused by the cache returning incorrect class bytes.

This problem should not occur under typical usage, but there are two situations in which it could happen:

- The classloader is using a SharedClassTokenHelper and the classes in the cache are out-of-date (dynamic updates are not supported with a SharedClassTokenHelper).
- Runtime bytecode modification is being used that is either not fully predictable in the modifications it does, or it is sharing a cache with another JVM that is doing different (or no) modifications. Regardless of the reason for the VerifyError, running in safemode (see "Using the safemode option" on page 261) should show if any bytecode in the cache is inconsistent with what the JVM is expecting. When you have determined the cause of the problem, destroy the cache, correct the cause of the problem, and try again.

Dealing with cache problems

The following list describes possible cache problems.

Cache is full

A full cache is not a problem; it just means that you have reached the limit of data that you can share. Nothing can be added or removed from that cache and so, if it contains a lot of out-of-date classes or classes that are not being used, you must destroy the cache and create a new one.

Cache is corrupt

In the unlikely event that a cache is corrupt, no classes can be added or read from the cache and a message is output to stderr. If the JVM detects that it is attaching to a corrupted cache, it will attempt to destroy the cache automatically. If the JVM cannot re-create the cache, it will continue to start only if **-Xshareclasses:nonfatal** is specified, otherwise it will exit. If a cache is corrupted during normal operation, all JVMs output the message and are forced to load all subsequent classes locally (not into the cache). The cache is designed to be resistant to crashes, so, if a JVM crash occurs during a cache update, the crash should not cause data to be corrupted.

`Could not create the Java virtual machine` message from utilities

This message does not mean that a failure has occurred. Because the cache utilities currently use the JVM launcher and they do not start a JVM, this message is always produced by the launcher after a utility has run. Because the JNI return code from the JVM indicates that a JVM did not start, it is an unavoidable message.

-Xscmx is not setting the cache size

You can set the cache size only when the cache is created because the size is fixed. Therefore, **-Xscmx** is ignored unless a new cache is being created. It does not imply that the size of an existing cache can be changed using the parameter.

Class sharing with OSGi ClassLoading framework

Eclipse releases after 3.0 use the OSGi ClassLoading framework, which cannot automatically share classes. A Class Sharing adapter has been written specifically for use with OSGi, which allows OSGi classloaders to access the class cache.

Using the JVMTI

JVMTI is a two-way interface that allows communication between the JVM and a native agent. It replaces the JVMDI and JVMPI interfaces.

JVMTI allows third parties to develop debugging, profiling, and monitoring tools for the JVM. The interface contains mechanisms for the agent to notify the JVM about the kinds of information it requires. The interface also provides a means of receiving the relevant notifications. Several agents can be attached to a JVM at any one time. A number of tools are based on this interface, such as Hyades, JProfiler, and Ariadna. These are third-party tools, therefore IBM cannot make any guarantees or recommendations regarding them. IBM does provide a simple profiling agent based on this interface, HPROF.

JVMTI agents can be loaded at startup using short or long forms of the command-line option:

```
-agentlib:<agent-lib-name>=<options>
```

or

```
-agentpath:<path-to-agent>=<options>
```

For example:

```
-agentlib:hprof=<options>
```

assumes that a folder containing `hprof.dll` is on the library path, or

```
-agentpath:C:\sdk\jre\bin\hprof.dll=<options>
```

For more information about JVMTI, see http://java.sun.com/javase/6/docs/technotes/guides/jvmti/.

For advice on porting JVMPI-based profilers to JVMTI, see http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition.

For a guide about writing a JVMTI agent, see http://java.sun.com/developer/technicalArticles/Programming/jvmti.

IBM JVMTI extensions

The IBM SDK provides extensions to the JVMTI. The sample shows you how to write a simple JVMTI agent that uses these extensions.

The IBM SDK extensions to JVMTI allow a JVMTI agent do the following tasks:

- Modify a dump.
- Modify a trace.
- Modify the logging configuration of the JVM.
- Initiate a JVM dump.

The definitions you need when you write a JVMTI agent are provided in the header files `jvmti.h` and `ibmjvmti.h`. These files are in `sdk/include`.

The sample JVMTI agent consists of two functions:

1. Agent_OnLoad()
2. DumpStartCallback()

Agent_OnLoad()

This function is called by the JVM when the agent is loaded at JVM startup, which allows the JVMTI agent to modify JVM behavior before initialization is complete. The sample agent obtains access to the JVMTI interface using the JNI Invocation API function GetEnv(). The agent calls the APIs GetExtensionEvents() and GetExtensionFunctions() to find the JVMTI extensions supported by the JVM. These APIs provide access to the list of extensions available in the jvmtiExtensionEventInfo and jvmtiExtensionFunctionInfo structures. The sample uses an extension event and an extension function in the following way:

The sample JVMTI agent searches for the extension event VmDumpStart in the list of jvmtiExtensionEventInfo structures, using the identifier `COM_IBM_VM_DUMP_START` provided in `ibmjvmti.h`. When the event is found, the JVMTI agent calls the JVMTI interface SetExtensionEventCallback() to enable the event, providing a function DumpStartCallback() that is called when the event is triggered.

Next, the sample JVMTI agent searches for the extension function SetVMDump in the list of jvmtiExtensionFunctionInfo structures, using the identifier `COM_IBM_SET_VM_DUMP` provided in `ibmjvmti.h`. The JVMTI agent calls the function using the jvmtiExtensionFunction pointer to set a JVM dump option `java:events=thrstart`. This option requests the JVM to trigger a javadump every time a VM thread is started.

DumpStartCallback()

This callback function issues a message when the associated extension event is called. In the sample code, DumpStartCallback() is used when the VmDumpStart event is triggered.

Compiling and running the sample JVMTI agent

Use this command to build the sample JVMTI agent on Windows:

```
cl /I<SDK_path>\include /MD /FetiSample.dll tiSample.c /link /DLL
```

where `<SDK_path>` is the path to your SDK installation.

Use this command to build the sample JVMTI agent on Linux:

```
gcc -I<SDK_path>/include -o libtiSample.so -shared tiSample.c
```

where `<SDK_path>` is the path to your SDK installation.

To run the sample JVMTI agent, use the command:

```
java -agentlib:tiSample -version
```

When the sample JVMTI agent loads, messages are generated. When the JVMTI agent initiates a javadump, the message `JVMDUMP010` is issued.

Sample JVMTI agent

A sample JVMTI agent, written in C/C++, using the IBM JVMTI extensions.

```
/*
 * tiSample.c
 *
 * Sample JVMTI agent to demonstratr the IBM JVMTI dump extensions
 */
```

```
#include "jvmti.h"
#include "ibmjvmti.h"

/* Forward declarations for JVMTI callback functions */
void JNICALL VMInitCallback(jvmtiEnv *jvmti_env, JNIEnv* jni_env, jthread thread);
void JNICALL DumpStartCallback(jvmtiEnv *jvmti_env, char* label, char* event, char* detail, ...);


/*
 * Agent_Onload()
 *
 * JVMTI agent initialisation function, invoked as agent is loaded by the JVM
 */
JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *jvm, char *options, void *reserved) {

  jvmtiEnv *jvmti = NULL;
 jvmtiError rc;
 jint extensionEventCount = 0;
 jvmtiExtensionEventInfo *extensionEvents = NULL;
 jint extensionFunctionCount = 0;
 jvmtiExtensionFunctionInfo *extensionFunctions = NULL;
 int i = 0, j = 0;

  printf("tiSample: Loading JVMTI sample agent\n");

    /* Get access to JVMTI */
    (*jvm)->GetEnv(jvm, (void **)&jvmti, JVMTI_VERSION_1_0);

    /* Look up all the JVMTI extension events and functions */
   (*jvmti)->GetExtensionEvents(jvmti, &extensionEventCount, &extensionEvents);
   (*jvmti)->GetExtensionFunctions(jvmti, &extensionFunctionCount, &extensionFunctions);

    printf("tiSample: Found %i JVMTI extension events, %i extension functions\n", extensionEventCount

   /* Find the JVMTI extension event we want */
   while (i++ &lt; extensionEvenCount) {

        if (strcmp(extensionEvents->id, COM_IBM_VM_DUMP_START) == 0) {
            /* Found the dump start extension event, now set up a callback for it */
            rc = (*jvmti)->SetExtensionEventCallback(jvmti, extensionEvents->extension_event_index, &
            printf("tiSample: Setting JVMTI event callback %s, rc=%i\n", COM_IBM_VM_DUMP_START, rc);
            break;
        }
        extensionEvents++; /* move on to the next extension event */
    }

    /* Find the JVMTI extension function we want */
    while (j++ < extensionFunctionCount) {
        jvmtiExtensionFunction function = extensionFunctions->func;

        if (strcmp(extensionFunctions->id, COM_IBM_SET_VM_DUMP) == 0) {
            /* Found the set dump extension function, now set a dump option to generate javadumps on
            rc = function(jvmti, "java:events=thrstart");
            printf("tiSample: Calling JVMTI extension %s, rc=%i\n", COM_IBM_SET_VM_DUMP, rc);
            break;
        }
        extensionFunctions++; /* move on to the next extension function */
     }

     return JNI_OK;

}


/*
 * DumpStartCallback()
 * JVMTI callback for dump start event (IBM JVMTI extension)  */
```

```
void JNICALL
DumpStartCallback(jvmtiEnv *jvmti_env, char* label, char* event, char* detail, ...) {
   printf("tiSample: Received JVMTI event callback, for event %s\n", event);
}
```

IBM JVMTI extensions - API reference

Reference information for the IBM SDK extensions to the JVMTI.

Setting JVM dump options

To set a JVM dump option use:

```
jvmtiError jvmtiSetVmDump(jvmtiEnv* jvmti_env, char* option)
```

The dump option is passed in as an ASCII character string. Use the same syntax as the **-Xdump** command-line option, with the initial **-Xdump:** omitted. See "Using the -Xdump option" on page 159.

When dumps are in progress, the dump configuration is locked, and calls to jvmtiSetVmDump() fail with a return value of `JVMTI_ERROR_NOT_AVAILABLE`.

Parameters:

jvmti_env: A pointer to the JVMTI environment.

option: The JVM dump option string.

Returns:

`JVMTI_ERROR_NONE`: Success.

`JVMTI_ERROR_NULL_POINTER`: The parameter **option** is null.

`JVMTI_ERROR_OUT_OF_MEMORY`: There is insufficient system memory to process the request.

`JVMTI_ERROR_INVALID_ENVIRONMENT`: The **jvmti_env** parameter is invalid.

`JVMTI_ERROR_WRONG_PHASE`: The extension has been called outside the JVMTI live phase.

`JVMTI_ERROR_NOT_AVAILABLE`: The dump configuration is locked because a dump is in progress.

`JVMTI_ERROR_ILLEGAL_ARGUMENT`: The parameter **option** contains an invalid **-Xdump** string.

Note: On z/OS, you might need to convert the option string from EBCDIC to ASCII before using this JVMTI extension function.

Querying JVM dump options

To query the current JVM dump options, use:

```
jvmtiError jvmtiQueryVmDump(jvmtiEnv* jvmti_env, jint buffer_size, void* options_buffer, jint* dat
```

This extension returns a set of dump option specifications as ASCII strings. The syntax of the option string is the same as the **-Xdump** command-line option, with the initial **-Xdump:** omitted. See "Using the -Xdump option" on page 159. The option strings are separated by newline characters. If the memory buffer is too small to contain the current JVM dump option strings, you can expect the following results:

- The error message `JVMTI_ERROR_ILLEGAL_ARGUMENT` is returned.
- The variable for `data_size_ptr` is set to the required buffer size.

Parameters:

jvmti_env: A pointer to the JVMTI environment.

buffer_size: The size of the supplied memory buffer in bytes.

options_buffer: A pointer to the supplied memory buffer.

data_size_ptr: A pointer to a variable, used to return the total size of the option strings.

Returns:

`JVMTI_ERROR_NONE`: Success

`JVMTI_ERROR_NULL_POINTER`: The **options_buffer** or **data_size_ptr** parameters are null.

`JVMTI_ERROR_OUT_OF_MEMORY`: There is insufficient system memory to process the request.

`JVMTI_ERROR_INVALID_ENVIRONMENT`: The **jvmti_env** parameter is invalid.

`JVMTI_ERROR_WRONG_PHASE`: The extension has been called outside the JVMTI live phase.

`JVMTI_ERROR_NOT_AVAILABLE`: The dump configuration is locked because a dump is in progress.

`JVMTI_ERROR_ILLEGAL_ARGUMENT`: The supplied memory buffer in **options_buffer** is too small.

Resetting JVM dump options

To reset the JVM dump options to the values at JVM initialization, use:

```
jvmtiError jvmtiResetVmDump(jvmtiEnv* jvmti_env)
```

Parameters:

jvmti_env: The JVMTI environment pointer.

Returns:

`JVMTI_ERROR_NONE`: Success.

`JVMTI_ERROR_OUT_OF_MEMORY`: There is insufficient system memory to process the request.

`JVMTI_ERROR_INVALID_ENVIRONMENT`: The **jvmti_env** parameter is invalid.

`JVMTI_ERROR_WRONG_PHASE`: The extension has been called outside the JVMTI live phase.

`JVMTI_ERROR_NOT_AVAILABLE`: The dump configuration is locked because a dump is in progress.

Triggering a JVM dump

To trigger a JVM dump, use:

```
jvmtiError jvmtiTriggerVmDump(jvmtiEnv* jvmti_env, char* option)
```

Choose the type of dump required by specifying an ASCII string that contains one of the supported dump agent types. See “Dump agents” on page 162. JVMTI events are provided at the start and end of the dump.

Parameters:

jvmti_env: A pointer to the JVMTI environment.

option: A pointer to the dump type string, which can be one of the following types:

- stack
- java
- system
- console
- tool
- heap
- snap
- ceedump (z/OS only)

Returns:

`JVMTI_ERROR_NONE`: Success.

`JVMTI_ERROR_NULL_POINTER`: The **option** parameter is null.

`JVMTI_ERROR_OUT_OF_MEMORY`: There is insufficient system memory to process the request.

`JVMTI_ERROR_INVALID_ENVIRONMENT`: The **jvmti_env** parameter is invalid.

`JVMTI_ERROR_WRONG_PHASE`: The extension has been called outside the JVMTI live phase.

`JVMTI_ERROR_NOT_AVAILABLE`: The dump configuration is locked because a dump is in progress.

Note: On z/OS, you might need to convert the option string from EBCDIC to ASCII before using this JVMTI extension function.

Setting JVM trace options

To set a JVM trace option, use:

```
jvmtiError jvmtiSetVmTrace(jvmtiEnv* jvmti_env, char* option)
```

The trace option is passed in as an ASCII character string. Use the same syntax as the **-Xtrace** command-line option, with the initial **-Xtrace:** omitted. See "Detailed descriptions of trace options" on page 214.

Parameters:

jvmti_env: JVMTI environment pointer.

option: Enter the JVM trace option string.

Returns:

`JVMTI_ERROR_NONE`: Success.

`JVMTI_ERROR_NULL_POINTER`: The **option** parameter is null.

`JVMTI_ERROR_OUT_OF_MEMORY`: There is insufficient system memory to process the request.

`JVMTI_ERROR_INVALID_ENVIRONMENT`: The **jvmti_env** parameter is invalid.

`JVMTI_ERROR_WRONG_PHASE`: The extension has been called outside the JVMTI live phase.

`JVMTI_ERROR_ILLEGAL_ARGUMENT`: The **option** parameter contains an invalid **-Xtrace** string.

Note: On z/OS, you might need to convert the option string from EBCDIC to ASCII before using this JVMTI extension function.

JVMTI event function - start dump

The following JVMTI event function is called when a JVM dump starts.

```
void JNICALL
VMDumpStart(jvmtiEnv *jvmti_env, JNIEnv* jni_env, char* label, char* event, char* detail)
```

The event function provides the dump file name and the name of the JVM event that triggered the dump. For more information about dump events, see "Dump events" on page 165.

Parameters:

jvmti_env: JVMTI environment pointer.

jni_env: JNI environment pointer for the thread on which the event occurred.

label: The dump file name, including directory path.

event: The extension event name, such as com.ibm.VmDumpStart.

detail: The dump event name.

Returns:

None

JVMTI event function - end dump

The following JVMTI event function is called when a JVM dump ends.

```
void JNICALL
VMDumpEnd(jvmtiEnv *jvmti_env, JNIEnv* jni_env, char* label, char* event, char* detail)
```

This event function provides the dump file name and the name of the JVM event that triggered the dump. For more information about dump events, see "Dump events" on page 165.

Parameters:

jvmti_env: JVMTI environment pointer.

jni_env: JNI environment pointer for the thread on which the event occurred.

label: The dump file name, including directory path.

event: The extension event name, such as com.ibm.VmDumpStart.

detail: The dump event name.

Returns:

None

Using the Diagnostic Tool Framework for Java

The Diagnostic Tool Framework for Java (DTFJ) is a Java application programming interface (API) from IBM used to support the building of Java diagnostics tools. DTFJ works with data from a system dump or a Javadump.

To work with a system dump, the dump must be processed by the jextract tool; see "Using the dump viewer" on page 194. The jextract tool produces metadata from

the dump, which allows the internal structure of the JVM to be analyzed. You must run jextract on the system that produced the dump.

To work with a Javadump, no additional processing is required.

The DTFJ API helps diagnostics tools access the following information:

- Memory locations stored in the dump (System dumps only)
- Relationships between memory locations and Java internals (System dumps only)
- Java threads running in the JVM
- Native threads held in the dump (System dumps only)
- Java classes and their classloaders that were present
- Java objects that were present in the heap (System dumps only)
- Java monitors and the objects and threads they are associated with
- Details of the workstation on which the dump was produced (System dumps only)
- Details of the Java version that was being used
- The command line that launched the JVM

If your DTFJ application requests information that is not available in the Javadump, the API will return null or throw a DataUnavailable exception. You might need to adapt DTFJ applications written to process system dumps to make them work with Javadumps.

This chapter describes DTFJ in:

- "Using the DTFJ interface"
- "DTFJ example application" on page 287

Using the DTFJ interface

To create applications that use DTFJ, you must use the DTFJ interface. Implementations of this interface have been written that work with WebSphere Real Time for RT Linux.

Figure 14 on page 286 illustrates the DTFJ interface. The starting point for working with a dump is to obtain an Image instance by using the ImageFactory class supplied with the concrete implementation of the API.

Working with a system dump

The following example shows how to work with a system dump.

```
import java.io.File;
import java.util.Iterator;
import java.io.IOException;

import com.ibm.dtfj.image.CorruptData;
import com.ibm.dtfj.image.Image;
import com.ibm.dtfj.image.ImageFactory;

public class DTFJEX1 {
    public static void main(String[] args) {
        Image image = null;
        if (args.length > 0) {
            File f = new File(args[0]);
            try {
                Class factoryClass = Class
```

```
                        .forName("com.ibm.dtfj.image.j9.ImageFactory");
                ImageFactory factory = (ImageFactory) factoryClass
                        .newInstance();
                image = factory.getImage(f);
            } catch (ClassNotFoundException e) {
                System.err.println("Could not find DTFJ factory class");
                e.printStackTrace(System.err);
            } catch (IllegalAccessException e) {
                System.err.println("IllegalAccessException for DTFJ factory class");
                e.printStackTrace(System.err);
            } catch (InstantiationException e) {
                System.err.println("Could not instantiate DTFJ factory class");
                e.printStackTrace(System.err);
            } catch (IOException e) {
                System.err.println("Could not find/use required file(s)");
                e.printStackTrace(System.err);
            }
        } else {
            System.err.println("No filename specified");
        }
        if (image == null) {
            return;
        }

        Iterator asIt = image.getAddressSpaces();
        int count = 0;
        while (asIt.hasNext()) {
            Object tempObj = asIt.next();
            if (tempObj instanceof CorruptData) {
                System.err.println("Address Space object is corrupt: "
                        + (CorruptData) tempObj);
            } else {
                count++;
            }
        }
        System.out.println("The number of address spaces is: " + count);
    }
}
```

In this example, the only section of code that ties the dump to a particular implementation of DTFJ is the generation of the factory class. Change the factory to use a different implementation.

The getImage() methods in ImageFactory expect one file, the `dumpfilename.zip` file produced by jextract (see see "Using the dump viewer" on page 194). If the getImage() methods are called with two files, they are interpreted as the dump itself and the `.xml` metadata file. If there is a problem with the file specified, an IOException is thrown by getImage() and can be caught and (in the example above) an appropriate message issued. If a missing file was passed to the above example, the following output is produced:

```
Could not find/use required file(s)
java.io.FileNotFoundException: core_file.xml (The system cannot find the file specified.)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:135)
        at com.ibm.dtfj.image.j9.ImageFactory.getImage(ImageFactory.java:47)
        at com.ibm.dtfj.image.j9.ImageFactory.getImage(ImageFactory.java:35)
        at DTFJEX1.main(DTFJEX1.java:23)
```

In the case above, the DTFJ implementation is expecting a dump file to exist. Different errors are caught if the file existed but was not recognized as a valid dump file.

Working with a Javadump

To work with a Javadump, change the factory class to com.ibm.dtfj.image.javacore.JCImageFactory and pass the Javadump file to the getImage() method.

```
import java.io.File;
import java.util.Iterator;
import java.io.IOException;

import com.ibm.dtfj.image.CorruptData;
import com.ibm.dtfj.image.Image;
import com.ibm.dtfj.image.ImageFactory;

public class DTFJEX2 {
    public static void main(String[] args) {
        Image image=null;

        if (args.length > 0) {
            File javacoreFile = new File(args[0]);

            try {
                Class factoryClass = Class.forName("com.ibm.dtfj.image.javacore.JCImageFactory");
                ImageFactory factory = (ImageFactory) factoryClass.newInstance();
                image = factory.getImage(javacoreFile);
            } catch .....
```

The rest of the example remains the same.

After you have obtained an Image instance, you can begin analyzing the dump. The Image instance is the second instance in the class hierarchy for DTFJ illustrated by the following diagram:

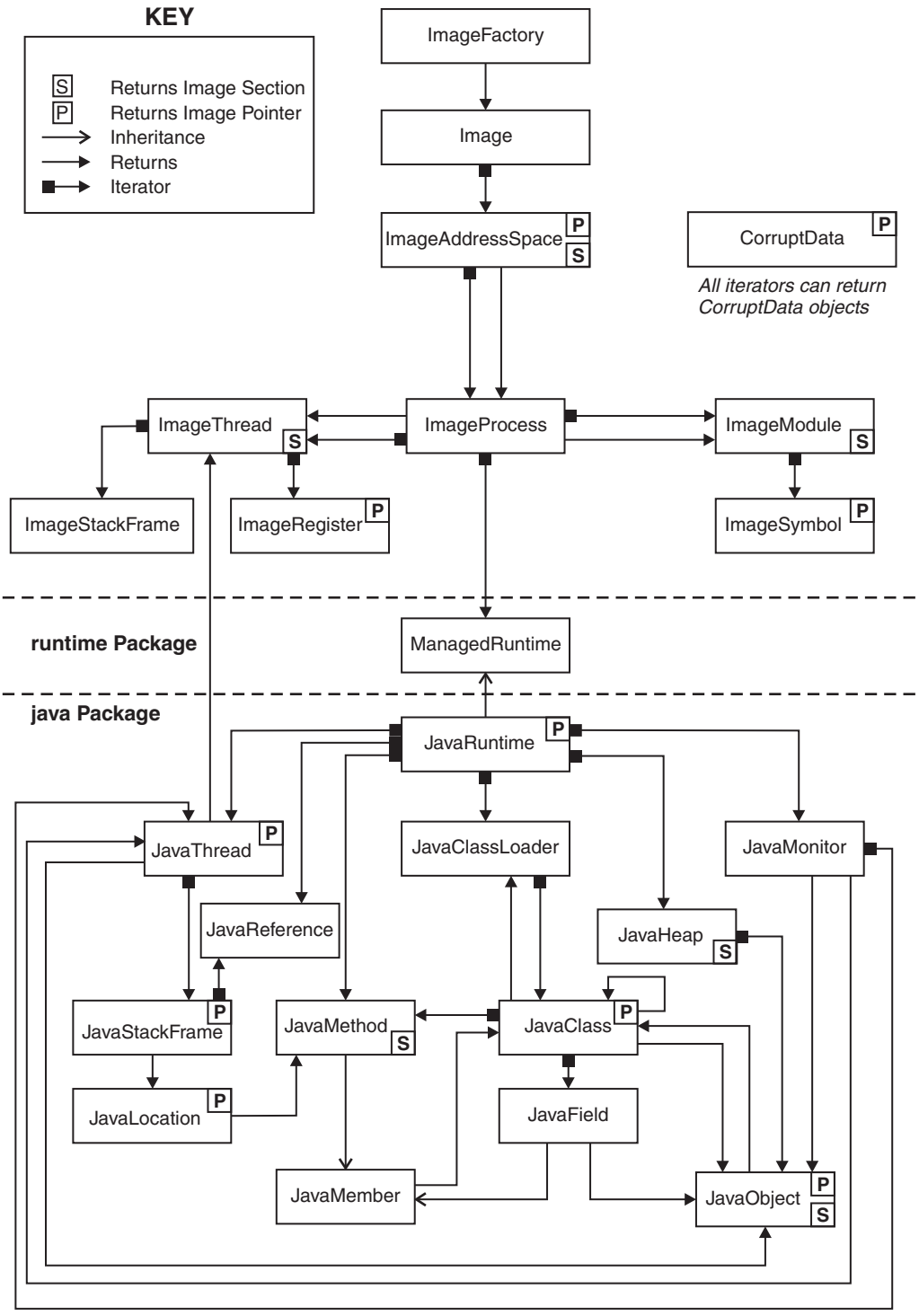


Figure 14. DTFJ interface diagram

The hierarchy displays some major points of DTFJ. Firstly, there is a separation between the Image (the dump, a sequence of bytes with different contents on different platforms) and the Java internal knowledge.

Some things to note from the diagram:

- The DTFJ interface is separated into two parts: classes with names that start with *Image* and classes with names that start with *Java*.
- *Image* and *Java* classes are linked using a ManagedRuntime (which is extended by JavaRuntime).
- An Image object contains one ImageAddressSpace object.
- An ImageAddressSpace object contains one ImageProcess object.
- Conceptually, you can apply the Image model to any program running with the ImageProcess, although for the purposes of this document discussion is limited to the IBM JVM implementations.
- There is a link from a JavaThread object to its corresponding ImageThread object. Use this link to find out about native code associated with a Java thread, for example JNI functions that have been called from Java.
- If a JavaThread was not running Java code when the dump was taken, the JavaThread object will have no JavaStackFrame objects. In these cases, use the link to the corresponding ImageThread object to find out what native code was running in that thread. This is typically the case with the JIT compilation thread and Garbage Collection threads.

DTFJ example application

This example is a fully working DTFJ application.

For clarity, this example does not perform full error checking when constructing the main Image object and does not perform CorruptData handling in all of the iterators. In a production environment, you use the techniques illustrated in the example in the "Using the DTFJ interface" on page 283.

In this example, the program iterates through every available Java thread and checks whether it is equal to any of the available image threads. When they are found to be equal, the program declares that it has, in this case, `"Found a match"`.

The example demonstrates:

- How to iterate down through the class hierarchy.
- How to handle CorruptData objects from the iterators.
- The use of the .equals method for testing equality between objects.

```
import java.io.File;
import java.util.Iterator;
import com.ibm.dtfj.image.CorruptData;
import com.ibm.dtfj.image.CorruptDataException;
import com.ibm.dtfj.image.DataUnavailable;
import com.ibm.dtfj.image.Image;
import com.ibm.dtfj.image.ImageAddressSpace;
import com.ibm.dtfj.image.ImageFactory;
import com.ibm.dtfj.image.ImageProcess;
import com.ibm.dtfj.java.JavaRuntime;
import com.ibm.dtfj.java.JavaThread;
import com.ibm.dtfj.image.ImageThread;

public class DTFJEX2
{
   public static void main( String[] args )
   {
      Image image = null;
      if ( args.length > 0 )
      {
         File f = new File( args[0] );
         try
         {
```

```
            Class factoryClass = Class
                  .forName( "com.ibm.dtfj.image.j9.ImageFactory" );
            ImageFactory factory = (ImageFactory) factoryClass.newInstance( );
            image = factory.getImage( f );
         }
         catch ( Exception ex )
         { /*
             * Should use the error handling as shown in DTFJEX1.
             */
            System.err.println( "Error in DTFJEX2" );
            ex.printStackTrace( System.err );
         }
      }
      else
      {
         System.err.println( "No filename specified" );
      }

      if ( null == image )
      {
         return;
      }

      MatchingThreads( image );
   }

   public static void MatchingThreads( Image image )
   {
      ImageThread imgThread = null;

      Iterator asIt = image.getAddressSpaces( );
      while ( asIt.hasNext( ) )
      {
         System.out.println( "Found ImageAddressSpace..." );

         ImageAddressSpace as = (ImageAddressSpace) asIt.next( );

         Iterator prIt = as.getProcesses( );

         while ( prIt.hasNext( ) )
         {
            System.out.println( "Found ImageProcess..." );

            ImageProcess process = (ImageProcess) prIt.next( );

            Iterator runTimesIt = process.getRuntimes( );
            while ( runTimesIt.hasNext( ) )
            {
               System.out.println( "Found Runtime..." );
               JavaRuntime javaRT = (JavaRuntime) runTimesIt.next( );

               Iterator javaThreadIt = javaRT.getThreads( );

               while ( javaThreadIt.hasNext( ) )
               {
                  Object tempObj = javaThreadIt.next( );
                  /*
                   * Should use CorruptData handling for all iterators
                   */
                  if ( tempObj instanceof CorruptData )
                  {
                     System.out.println( "We have some corrupt data" );
                  }
                  else
                  {
                     JavaThread javaThread = (JavaThread) tempObj;
                     System.out.println( "Found JavaThread..." );
```

```
                    try
                    {
                       imgThread = (ImageThread) javaThread.getImageThread( );

                       // Now we have a Java thread we can iterator
                       // through the image threads
                       Iterator imgThreadIt = process.getThreads( );

                       while ( imgThreadIt.hasNext( ) )
                       {
                          ImageThread imgThread2 = (ImageThread) imgThreadIt
                                .next( );
                          if ( imgThread.equals( imgThread2 ) )
                          {
                             System.out.println( "Found a match:" );
                             System.out.println( "\tjavaThread "
                                   + javaThread.getName( )
                                   + " is the same as " + imgThread2.getID( ) );
                          }
                       }
                    }
                    catch ( CorruptDataException e )
                    {
                       System.err.println( "ImageThread was corrupt: "
                             + e.getMessage( ) );
                    }
                    catch ( DataUnavailable e )
                    {
                       System.out.println( "DataUnavailable: "
                             + e.getMessage( ) );
                    }
                 }
              }
           }
        }
     }
  }
}
```

Many DTFJ applications will follow similar models.

Using the IBM Monitoring and Diagnostic Tools for Java - Health Center

The IBM Monitoring and Diagnostic Tools for Java - Health Center is a diagnostic tool for monitoring the status of a running Java Virtual Machine (JVM).

Introduction

The IBM Monitoring and Diagnostic Tools for Java - Health Center is a diagnostic tool for monitoring the status of a running Java Virtual Machine (JVM). The Heath Center uses a small amount of processor time and memory, and can open some log and trace files for analysis.

The tool is provided in two parts:

- The Health Center agent that collects data from a running application. To download the agent, see: “Installing the Health Center agent” on page 293.
- An Eclipse-based client that connects to the agent. The client interprets the data and provides recommendations to improve the performance of the monitored application.

The Health Center can be used to monitor Java applications, where the applications use one of the following JVMs:

- Java 6 SR1 and later
- Java 5.0 SR8 and later
- WebSphere Real Time for Linux V2 SR2 with APAR IZ61672 and later service refreshes

The Health Center allows monitoring of applications using WebSphere Real Time for Linux. However, monitoring production environments based on WebSphere Real Time for Linux is not recommended. The reason is that trace output functionality causes Health Center to generate log files that might consume unlimited amounts of disk space.

The Health Center is provided as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see: http://www.ibm.com/developerworks/java/jdk/tools/healthcenter/.

When the Health Center client starts up, you initially see a connection wizard. You can then:

- After installing the Health Center agent and enabling a Java application for monitoring, make a connection to the running application. See "Monitoring a running Java application" on page 292 for more information.
- Open a log file from disk by canceling the wizard. See "Opening files from disk" on page 301 for more information.

The Health Center client is split into subsystems, each representing a component of the JVM. The following subsystems are available:

- *Classes*: Information about classes being loaded
- *Environment*: Details of the configuration and system of the monitored application
- *Garbage collection*: Information about the Java heap and pause times
- *I/O*: Information about I/O activities that take place.
- *Locking*: Information about contention on inflated locks
- *Memory*: Information about the native memory usage
- *Profiling*: Provides a sampling profile of Java methods including call paths
- *WebSphere Real Time for Linux*: Information about real-time applications

Subsystems are represented as Eclipse perspectives. The first subsystem you see is the Status perspective, listing the subsystems and their overall status. When you connect to a running application or open a file (see "Opening files from disk" on page 301 for more information), subsystems with data available become links and any recommendations are displayed. The Health Center updates the displayed data and recommendations every 10 seconds. Switch to the subsystem perspectives using the links or the toolbar icons. You can return to the Status perspective using the furthest left toolbar icon.

You can send bug reports, feature requests, and feedback through your IBM representative, or you can post feedback or ask questions on the Health Center forum: http://www.ibm.com/developerworks/forums/forum.jspa?forumID=1461.

For more information about Health Center, including late-breaking news, see: http://www.ibm.com/developerworks/java/jdk/tools/healthcenter/release_notes.html.

Platform requirements

The Health Center client and Health Center agent have unique platform requirements. The functionality available with the agent depends on the level of Java Runtime Environment (JRE) you are using.

Platform requirements for the client

The Health Center client requires either the Microsoft® Windows or Linux x86 operating system using the supplied JRE. The client is Eclipse RCP-based; the minimum operating system requirements for the client are the same as the Eclipse RCP project, see http://www.eclipse.org/documentation/.

Platform requirements for the agent

The application that you want to monitor requires a minimum level of JRE with a Health Center agent installed. Later levels of JRE provide more Health Center function; the table shows at which JRE service refresh each function becomes available.

Some JRE levels come with an agent installed by default. To enable more function, install a later, updated, agent. See "Installing the Health Center agent" on page 293 for more information. The level of function provided by default and updated agents is described in the following table.

To use Health Center on a production system, run Java 5 JRE SR10 or later, or run Java 6 JRE SR5 or later.

Using the Health Center in production environments

The Health Center has a minimal affect on the system being monitored. However, it is not suitable for production use on Java 5 JRE before SR10, or Java 6 JRE before SR5.

Java version	Function with default agent	Function with updated agent	Suitable for production use	Command-line options to enable agent
Java 5 SR8	No agent included	Profiling, Garbage collection, locking, classes, environment, memory	No	`-agentlib:healthcenter -Xtrace:output=perfmon.%p.out`
Java 5 SR9	Profiling, Garbage collection, locking	Profiling, Garbage collection, locking, classes, environment, memory	No	`-agentlib:healthcenter -Xtrace:output=perfmon.%p.out`
Java 5 SR10 or later	Profiling, Garbage collection, locking, classes, environment	Profiling, Garbage collection, locking, classes, environment, memory, large object allocation, IO	Yes	`-Xhealthcenter`

Java version	Function with default agent	Function with updated agent	Suitable for production use	Command-line options to enable agent
Java 6 SR1	No agent included	Profiling, Garbage collection, locking, classes, environment, memory	No	`-agentlib:healthcenter -Xtrace:output=perfmon.%p.out`
Java 6 SR2	No agent included	Profiling, Garbage collection, locking, classes, environment, memory	No	`-agentlib:healthcenter -Xtrace:output=perfmon.%p.out`
Java 6 SR3	Profiling, Garbage collection, locking, classes	Profiling, Garbage collection, locking, classes, environment, memory	No	`-agentlib:healthcenter -Xtrace:output=perfmon.%p.out`
Java 6 SR4	Profiling, Garbage collection, locking, classes	Profiling, Garbage collection, locking, classes, environment, memory	No	`-agentlib:healthcenter -Xtrace:output=perfmon.%p.out`
Java 6 SR5 or later	Profiling, Garbage collection, locking, classes, environment	Profiling, Garbage collection, locking, classes, environment, memory, large object allocation, IO	Yes	`-Xhealthcenter`
WebSphere Real Time for Linux V2 SR2 with APAR IZ61672	Profiling, Garbage collection, locking, classes, environment	Profiling, Garbage collection, locking, classes, environment, memory	No	`-agentlib:healthcenter -Xtrace:output=perfmon.%p.out`
WebSphere Real Time for Linux V2 SR3 or later service refreshes	Profiling, Garbage collection, locking, classes, environment, WebSphere Real Time	Profiling, Garbage collection, locking, classes, environment, memory, WebSphere Real Time, IO	Yes	`-Xhealthcenter`

Monitoring a running Java application

Use the Health Center to connect to, and monitor, a Java application.

To monitor a running Java application, you must:

1. Install the Health Center agent into the IBM Java Virtual Machine (JVM) for the Java application. See “Installing the Health Center agent” on page 293
2. Start the Java application with the agent enabled. See “Starting a Java application with the Health Center agent enabled” on page 294 for more information.

3. Connect to the Java application using the Health Center client. See "Connecting to a Java application using the Health Center client" on page 295 for more information.

To learn more about the data that the Health Center client displays, see "Data available on connection to a running Java application" on page 299.

Installing the Health Center agent

Download and install the correct agent package for the Java version you are using.

Procedure

Some IBM Java Runtime Environments (JREs) already have a Health Center agent installed. However, you should still install the agent using this procedure to ensure that the latest updates are included.

1. Download the agent package by clicking the link corresponding to the Java version you are running:

   Note: This might not be the same as the operating system you are running. For example, you might be running a 32-bit Java on a Windows 64-bit system; in this case you should download the Windows 32-bit agent package.

   - Windows x86 32-bit
   - Windows x86 64-bit
   - Linux x86 32-bit
   - Linux x86 64-bit
   - Linux s390 31-bit
   - Linux s390 64-bit
   - Linux ppc 32-bit
   - Linux ppc 64-bit
   - AIX ppc 32-bit
   - AIX ppc 64-bit
   - z/OS 31-bit
   - z/OS 64-bit
2. Install the agent.
   - **Installing on Microsoft Windows, AIX and Linux**:
     a. You must download and extract the agent package into a specific directory of the JRE that you are using to start your application. This directory is the **parent** directory of the `jre` directory. For example, on Microsoft Windows, if your JRE is in the `C:\Program Files\IBM\Java60\jre` directory, extract the contents of the Windows x86 32-bit agent package into `C:\Program Files\IBM\Java60`
     b. When extracted, you see a healthcenter.jar file in the `jre\lib\ext` directory. Using the example in step **a.**, your healthcenter.jar is in `C:\Program Files\IBM\Java60\jre\lib\ext`.
   - **Installing on z/OS**:
     a. Unpack the z/OS agent package (a pax file) into the SDK directory using the command **pax -ppx -rf**. For example:

     ```
     W0 /u/user/J5.0: pax -ppx -rf ../mz31.pax
     ```

     The agent files are unpacked into the Java SDK directory. For example, for Java 5.0, this directory is similar to `/u/user/J5.0`.

Starting a Java application with the Health Center agent enabled

There are two ways to activate the Health Center agent when your Java application is started. There are additional considerations for specific WebSphere or Rational® products.

Prerequisite

The Health Center agent must be installed. See "Installing the Health Center agent" on page 293 for more information.

Procedure

To monitor an application, the Health Center agent must be enabled when the JVM is started. There are two ways to do this:

1. Start Java from the command line using the appropriate Health Center option, which is described in full in the "Platform requirements" on page 291 section.

   For example, with Java 5 SR9 and earlier, or Java 6 SR4 and earlier, use:

   ```
   java -agentlib:healthcenter -Xtrace:output=perfmon.out -classpath my/class/path.jar MyMainClass
   ```

   For Java 5 SR10 and later, or Java 6 SR5 and later, use:

   ```
   java -Xhealthcenter -classpath my/class/path.jar MyMainClass
   ```

2. Use the IBM_JAVA_OPTIONS environment variable to set the Health Center agent option before running your Java command. For example, on Microsoft Windows, with Java 5 SR9 and earlier, or Java 6 SR4 and earlier, enter the following command:

   ```
   set IBM_JAVA_OPTIONS="-agentlib:healthcenter -Xtrace:output=perfmon.out"
   ```

   For Java 5 SR10 and later, or Java 6 SR5 and later, type:

   ```
   set IBM_JAVA_OPTIONS="-Xhealthcenter"
   ```

   When the option is set you can start Java.

After the JVM is started with the agent enabled, you see a message detailing the port for the Health Center agent. For example:

```
05-Mar-2009 09:49:57 com.ibm.java.diagnostics.healthcenter.agent.mbean.HCLaunchMBean startAgent
INFO: Health Center agent started on port 1972.
```

The port number is also written to the `healthcenter.<pid>.log` file in the users temporary directory. The `<pid>` is the process ID for the agent that is listening on that port.

To enable the Health Center agent when a JVM is started in a WebSphere or Rational product environment, see "Configuring WebSphere or Rational product environments" on page 296.

Changing the listening port

By default, the Health Center agent uses port 1972 for its communications. If it cannot use port 1972, it increments the port number and tries again, for up to 100 attempts. You can override the first port number that the agent tries to use.

If you are using a JVM level that provides the **-Xhealthcenter** option (described in the "Platform requirements" on page 291 section), you can specify the port as a command-line option. For example:

```
java -Xhealthcenter:port=<port_number> HelloWorld
```

Otherwise, use the `com.ibm.java.diagnostics.healthcenter.agent.port` command-line option. For example:

```
java -agentlib:healthcenter -Xtrace:output=perfmon.out -Dcom.ibm.java.diagnostics.healthcenter.age
```

To change the port permanently, edit the following line in the `healthcenter.properties` file.

```
com.ibm.java.diagnostics.healthcenter.agent.port
```

This file is in the `jre/lib` directory of the JVM containing the agent.

Starting the Health Center agent without a client connection

From Health Center V1.2, you can start the agent without a client connection in place. The agent waits for a client connection with no impact on the application that is running. When the client connects to the agent, data collection starts. To configure an agent to start in this mode, edit the following line in the `healthcenter.properties` file, and change the value to off.

```
com.ibm.java.diagnostics.healthcenter.data.collection.level
```

This file is in the `jre/lib` directory of the JVM containing the agent.

For more information about troubleshooting problems with the Health Center agent, see "Cannot connect to an application" on page 320.

Connecting to a Java application using the Health Center client

You can connect the Health Center client to a Java application that you want to monitor.

Prerequisite

The JVM in which the Java application is running must have the Health Center agent installed and active. See "Installing the Health Center agent" on page 293 and "Starting a Java application with the Health Center agent enabled" on page 294 for more information.

Procedure

To connect the Health Center client to a Java application:

1. Select **New Connection** from the **File** menu of an open Health Center client, or start the client. A connection wizard is displayed.
2. Ensure that you have enabled your application for monitoring then click **Next**.
3. Specify the host name and port number. The Health Center makes a connection using these details. The Health Center can scan for open ports on a machine that might have agents waiting for a connection. This behavior is enabled by the `Scan next 100 ports for available connections` option.
4. If you require authentication, select the appropriate option and enter a user name and password.
5. Click **Next** to find available connections in the host and port range specified. Select a connection from the list of connections found.
6. Click **Finish** to connect to the selected host and port.

After the wizard finishes, the Health Center attempts to connect to the host name and port that you specified. A message dialog box notifies you if authentication is required.

If you cannot connect to the application, see the troubleshooting topic: “Cannot connect to an application” on page 320.

Connecting to a Java application using authentication:

Authentication provides a secure way of accessing your Java application through the Health Center agent.

Using authentication: agent setup

For the Health Center agent, authentication is configured using files on disk. The agent requires an authentication file to configure the user name and password, and an authorization file to configure access for that user name. The authentication file contains the user name and password, separated by a space. The authorization file contains the user name and the word `readwrite`, separated by a space. Here is a sample authentication file for `authentication.txt`:

```
myuser mypassw0rd
```

The associated `authorization.txt` file is similar to:

```
myuser readwrite
```

Note: Use only alphabetic characters for the user name and password. Do not use spaces or symbols.

You can choose the files to use by configuring the following command-line options:

- com.ibm.java.diagnostics.healthcenter.agent.authentication.file
- com.ibm.java.diagnostics.healthcenter.agent.authorization.file

For example:

```
java -agentlib:healthcenter -Xtrace:output=perfmon.out \
-Dcom.ibm.java.diagnostics.healthcenter.agent.authentication.file=/home/user/authentication.txt \
-Dcom.ibm.java.diagnostics.healthcenter.agent.authorization.file=/home/user/authorization.txt \
MyClassName
```

Ensure that you configure the permissions on the authentication file so that only authorized users can see the password information it holds.

Using authentication: client setup

To use authentication with the Health Center client, tick the authentication option on the wizard page and enter the user name and password stored in the `authentication.txt` file.

Configuring WebSphere or Rational product environments

Learn how to enable the Health Center agent for Java applications that run in specific environments.

Prerequisite

The Health Center agent must be installed. See “Installing the Health Center agent” on page 293 for more information about how to install the Health Center agent.

Procedure

Before using the Health Center client to connect to a Health Center agent running in specific WebSphere or Rational environments, the Health Center agent must be started. The steps you need to follow to start the agent can be different depending on the products you are using.

Configuring WebSphere Application Server environments:

To enable Health Center monitoring in a WebSphere Application Server environment, use the administration console to change the configuration.

Prerequisite

The Health Center agent must be installed. See "Installing the Health Center agent" on page 293 for more information about how to install the Health Center agent before you configure WebSphere Application Server.

Procedure

To enable the Health Center for use with WebSphere Application Server:

1. Select **Servers** ->**Server Types** -> **WebSphere application servers**.
2. Select the server name and then select **Java and Process Management** -> **Process definition** -> **Java Virtual Machine** -> **Generic JVM Arguments**.
3. Enter the correct string for your Java Virtual Machine (JVM) version.

   On UNIX system based platforms:

   - For Java 5 SR9 and earlier or Java 6 SR4 and earlier, use:

     ```
     -agentlib:healthcenter -Xtrace:output=/tmp/perfmon.%p.out
     ```

   - For Java 5 SR10 and later, or Java 6 SR5 and later, use:

     ```
     -Xhealthcenter
     ```

   On Microsoft Windows:

   - For Java 5 SR9 and earlier or Java 6 SR4 and earlier, use:

     ```
     -agentlib:healthcenter -Xtrace:output=C:\temp\perfmon.%p.out
     ```

   - For Java 5 SR10 and later, or Java 6 SR5 and later, use:

     ```
     -Xhealthcenter
     ```

4. Apply the changes and save the settings at the top of the page.
5. Restart the JVM.

   On UNIX system based platforms, the `perfmon*.out` files are in `/tmp`.

   On Microsoft Windows, the `perfmon*.out` files are in your temporary directory.
6. Connect to the server from the Health Center client. The default port number is 1972. For more information about connecting, see "Connecting to a Java application using the Health Center client" on page 295.

Configuring WebSphere Integration Developer environments:

To enable Health Center monitoring of a WebSphere Application Server test environment in WebSphere Integration Developer, use the administration console to change the configuration.

Prerequisite

The Health Center agent must be installed. See "Installing the Health Center agent" on page 293 for more information about how to install the agent before you start the WebSphere Application Server test environment with monitoring enabled.

Procedure

To enable the Health Center for use with the WebSphere Application Server test environment in WebSphere Integration Developer.

1. Locate the Java Runtime Environment (JRE) directory for the test environment runtime that you want to monitor with the Health Center. For WebSphere Application Server v6.1 in WebSphere Integration Developer on a Microsoft Windows system, this directory is typically `C:\Program Files\IBM\SDP70\runtimes\base_v61\java\jre`.
2. Copy the agent files into this directory.
3. From the WebSphere Integration Developer console, select **Servers** and select the test environment runtime that you want to start.
4. Use the right mouse button to display a list of actions and select **Start**.
5. From the WebSphere Integration Developer console, select **Servers** and select the test environment runtime again.
6. Use the right mouse button to display a list of actions and select **Run administrative console**.
7. When the admin console has started, locate and update the generic Java Virtual Machine (JVM) arguments field by expanding **Servers** → **Application servers** → **server1**.
8. Next, expand **Java and Process Management** → **Process Management** → **Process Definition** → **Java Virtual Machine**.
9. Add the correct string for your JVM version to the end of the **Generic JVM arguments** field.
   - For Java 5 SR9 and earlier or Java 6 SR4 and earlier, use:
     `-agentlib:healthcenter -Xtrace:output=perfmon.%p.out`
   - For Java 5 SR10 and later, or Java 6 SR5 and later, use:
     `-Xhealthcenter`
10. Select **Apply** and **Save**.
11. From the WebSphere Integration Developer console, select **Servers** and restart the test environment runtime. Do this by using the right mouse button to display a list of actions and select **Restart** → **Start**.
12. Start the Health Center client.
13. Connect to the system that is running the WebSphere Integration Developer test environment runtime. For more information about connecting, see "Connecting to a Java application using the Health Center client" on page 295.
14. You can check that you have connected to the correct test environment from the Health Center GUI. Select **Environment** to see the environment perspective and select **Java Virtual Machine**. The `Java Home` value is the same as the JRE directory that you located in the first step.

Configuring Rational Application Developer environments:

To enable Health Center monitoring of a WebSphere Application Server test environment in Rational Application Developer, use the administration console to change the configuration.

Prerequisite

The Health Center agent must be installed. See "Installing the Health Center agent" on page 293 for more information about how to install the agent before you start the WebSphere Application Server test environment with monitoring enabled.

Procedure

To enable the Health Center for use with the WebSphere Application Server test environment in Rational Application Developer:

1. Locate the Java Runtime Environment (JRE) directory for the test environment runtime that you want to monitor with the Health Center. For WebSphere Application Server v6.1 in Rational Application Developer 7.0 on a Microsoft Windows system, this directory is typically `C:\Program Files\IBM\SDP70\runtimes\base_v61\java\jre`.
2. Copy the agent files into this directory.
3. From the Rational Application Developer console, select **Servers** and select the test environment runtime that you want to start.
4. Use the right mouse button to display a list of actions and select **Start**.
5. From the Rational Application Developer console, select **Servers** and select the test environment runtime again.
6. Use the right mouse button to display a list of actions and select **Run administrative console**.
7. When the admin console has started, locate and update the generic Java Virtual Machine (JVM) arguments field by expanding **Servers** → **Application servers** → **server1**.
8. Next, expand **Java and Process Management** → **Process Management** → **Process Definition** → **Java Virtual Machine**.
9. Add the correct string for your JVM version to the end of the **Generic JVM arguments** field.
   - For Java 5 SR9 and earlier or Java 6 SR4 and earlier, use:
     ```
     -agentlib:healthcenter -Xtrace:output=perfmon.%p.out
     ```
   - For Java 5 SR10 and later, or Java 6 SR5 and later, use:
     ```
     -Xhealthcenter
     ```
10. Select **Apply** and **Save**.
11. From the Rational Application Developer console, select **Servers** and restart the test environment runtime. Do this by using the right mouse button to display a list of actions and select **Restart** → **Start**.
12. Start the Health Center client.
13. Connect to the system that is running the Rational Application Developer test environment runtime. For more information about connecting, see "Connecting to a Java application using the Health Center client" on page 295.
14. You can check that you have connected to the correct test environment from the Health Center GUI. Select **Environment** to see the environment perspective and select **Java Virtual Machine**. The `Java Home` value is the same as the JRE directory that you located in the first step.

Data available on connection to a running Java application

When connecting the Health Center to a running Java application, the data available for the client to display can vary for a number of reasons.

The data available to the Health Center client on connection to a live source varies depending on the following conditions:

- If this is the first Health Center client that has connected to a particular live source since it was started.
- The version of the Java Virtual Machine (JVM) being monitored.

- The version of the Health Center agent used (see the section on platform requirements for the agent in: "Platform requirements" on page 291).

Health Center clients consume data from the system to which they are connected. Therefore, after the first time that a Health Center client connects to a particular live source, subsequent Health Center client connections to the same source will not have access to data used by the first Health Center client connection.

For example:

1. Health Center client A connects to live source L.
2. Health Center client A uses some data from live source L and then disconnects.
3. Health Center client B connects to live source L.

When Health Center client B connects, the data that was used by client A is no longer available, so client B can access only the data that client A did not use.

Data available on first Health Center client connections to a live source

Java version	Available data
Java 5 SR9 and earlier, or Java 6 SR4 and earlier	Data from the time when the live source was started until the current time.
Java 5 SR10 and later, or Java 6 SR5 and later	As much historical data as fits in the Health Center agents buffer, up to the current time.

Data available on subsequent Health Center client connection to a live source

Java version	Available data
Java 5 SR9 and earlier, or Java 6 SR4 and earlier	All data from the time when the live source was started until the current time, which has not already been used by a Health Center client. **Restriction:** Method names are available only for classes loaded since the previous Health Center client was disconnected.
Java 5 SR10 and later, or Java 6 SR5 and later	As much historical data as fits in the Health Center agents buffer, up to the current time, which has not already been used by a Health Center client. **Note:** In the profiling perspective, some method names might not display immediately.

Controlling the amount of data generated

How to control the amount of generated data, in order to prevent loss of data.

If an application generates more data than Health Center can process, it is possible that Health Center might lose some data. If data loss occurs, you see a message about dropped data points in the agent connection view.

You can reduce the likelihood of losing data by turning off individual perspectives if you are not interested in the data they display. If a perspective is turned off, data for that perspective is no longer generated and sent to the Health Center client.

To turn off a perspective, use the preferences option under **Subsystem Enablement**.

Saving data

Health Center can save the data that it is currently analyzing to a `.hcd` file on the hard disk.

The `.hcd` file can be opened by the Health Center at a later date without the need for a live connection. The file contains data showing what the system looked like at the time the data was saved. The file does not need to be opened by the Health Center that created it. The file can be passed to another installation of the same version for analysis. For more information, see "Opening files from disk."

Saving data to disk

To save data to disk, select **File → Save Data**.

You are prompted to enter a file name and location for saving the data.

The amount of disk space used to export data is configurable. By default, the disk space is 300 MB. This means that only the most recent 300 MB of data read by the Health Center is available to save. The quantity of information produced by the monitored application determines the time duration included in the 300 MB of data. For example, an application producing little information might record the last 10 hours of trace data in 300 MB of space. An application producing much information might only record the last 10 minutes of data.

To change the amount of disk space used to save data, use the disk space management option under **Preferences → Data Storage Settings**. To save all the data read by the Health Center, clear this option.

CAUTION:
If you remove the limit on the file size, the file might grow until you run out of disk space.

All the data currently available is exported. If you cropped the data displayed by dragging to select only a particular time interval, then the cropping settings are lost when you import the file. If you have enabled **Sliding window truncation**, then data outside of the sliding window you selected is exported if the data has not yet been removed from the disk. The exported data is available when you import the file later.

Opening files from disk

When Health Center monitors a Java application, data is stored to disk.

Health Center can analyze log files gathered from an earlier invocation of a Java Virtual Machine (JVM), without making a live connection. To open log files from disk, cancel the connection wizard that appears when the client is started.

Opening saved data files

If you saved data previously, you can load the data back into the Health Center. Saved data files have a `.hcd` file extension. Older releases of the Health Center stored data in files with a `.zip` file extension.

To load saved data, select **File → Open File**.

Select the name of the `.hcd` or `.zip` file containing the data to load. Depending on the quantity of data to import, it might take a while for the Health Center to process the files. When the import finishes, the Health Center displays the information.

Opening log files

For Java 5 SR9 and earlier or Java 6 SR4 and earlier, the Health Center agent stores data to disk in the `perfmon.out` file. To open a log file, select **File → Open File**.

The Health Center can parse trace files containing garbage collection information or profiling information. These trace files are created automatically by the Health Center agent when enabled with the **-agentlib:healthcenter** and **-Xtrace:output=perfmon.out** command-line options.

When a file is opened, the Health Center attempts to parse it and analyze the parsed data. On completion of the analysis, the status view and perspective are updated to show the available information.

JVM subsystems for which data is available are linked. For further information, you can click the available links, including **Profiling**, **Classes**, **Locking** and **Garbage Collection**.

Classes perspective

Class loading might be a cause of failures or performance problems.

Class loading often causes difficulties for application developers. It might prevent a class from functioning correctly; for example, being unable to resolve a class or loading an incorrect version of a class. Performance problems during class loading can also occur; for example, the application might pause when a new class is loaded and the pause triggers the loading of other classes; or classes might be constantly being loaded.

Be aware that class loading might cause memory usage problems. When a class is loaded, it uses the native heap, which is released only when the class loader that loaded it is garbage collected. If a class loader does not become eligible for garbage collection when expected, native heap is not freed appropriately.

If you see an OutOfMemory error, it is likely that more classes have been loaded over time than are unloaded, and the available memory on the heap has decreased.

Using the classes perspective

The classes perspective displays the density of class loading over time, which classes were loaded at which time, and whether a class was loaded from the class sharing cache.

The class loading timeline

The class loading graph gives a visual indication of how much class loading occurred in your application over time. Use the graph to identify points in time at which classes were loading at a rate you did not expect.

The classes table

The classes table gives a more detailed view of which classes have been loaded at which times. This table also indicates whether the class was loaded from the shared classes cache.

Column heading	Description
Time loaded	The time, measured from Java Virtual Machine (JVM) start time, when the class was loaded.
Shared cache	Whether the class was loaded from the shared classes cache. Not all classes can be cached.
Classname	The full name of the loaded class.

Filtering the classes table

Use the text box above the table to filter the output of the classes table. For further information about filtering, see the filtering help topic.

Viewing data for a particular time period

You can select the time interval for displaying data, and making recommendations, by using cropping. For further information about cropping, see "Cropping data" on page 325.

Related concepts

"Cropping data" on page 325
You can change the time period for which data is displayed and on which recommendations are based.

Class references

Links to some websites for more information about classes.

You can analyze and understand Java class loading problems through the following links:

- *Class loading*: class loading is described in the class loading section of the Java Diagnostics Guide.
- *Class data sharing*: class data sharing is described in the class data sharing section of the Java Diagnostics Guide.
- *Java classes and class loading*: a basic introduction to class files and class loaders.
- *Class sharing*: an introduction to the shared classes feature available in IBM JVMs to reduce memory footprint and increase startup performance.

Environment perspective

Areas monitored by the environment perspective.

The environment perspective shows system and configuration information about the monitored Java Virtual Machine (JVM), including:

- Version information for the JVM
- Operating system and architecture information for the monitored system
- Process ID
- All system properties
- All environment variables

This information can be useful in confirming that the intended JVM is being monitored. You can use this information to help diagnose some types of problems.

The Health Center identifies JVM parameters that might adversely affect system performance, stability, and serviceability. If any of these parameters are detected, a warning is displayed.

Environment references

Links to some websites for more information about environment.

You can analyze and understand Java environment problems through the following links:

- Nonstandard command-line options provides a list of all the IBM -X options supported by IBM.
- Revelations on Java signal handling and termination discusses how the Java Virtual Machine (JVM) handles signals and how to write signal handlers.

Garbage collection perspective

Identify memory leaks and review suggested tuning parameters.

Garbage collection is a system of automatic memory management. Memory that has been dynamically allocated but that is no longer in use is reclaimed without intervention by the application. Garbage collection solves the problem of determining object liveness by freeing memory only when it becomes unreachable.

Garbage collection offers many benefits in terms of application robustness and performance. The Java Virtual Machine (JVM) auto-tunes garbage collection but explicit tuning can improve performance or bring application behavior in line with quality of service requirements. You can also use garbage collection to identify applications that are not running properly. Excessive memory consumption can have a significant performance affect. A memory leak can cause an application to fail.

The Health Center attempts to suggest tuning parameters and identify memory leaks.

Enabling the garbage collection perspective

Enable the garbage collection perspective:

1. Connect to a JVM running the Health Center agent.
2. Open the binary trace log from a JVM running the Health Center agent.

More detailed garbage collection information is available from Java 6 than from Java 5.

Using the garbage collection perspective

The heap usage, pause times, summary table, and tuning recommendation sections in the Health Center garbage collection perspective.

The Health Center garbage collection perspective has the following sections:

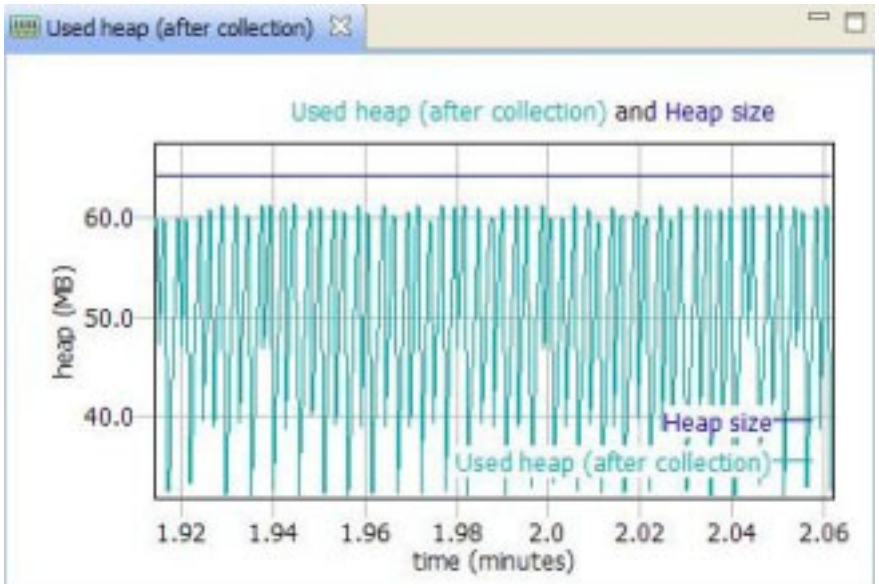
- A graph of heap usage.
- A graph of pause times.
- A summary table of important GC metrics.
- Tuning recommendations.

- A table showing the call stacks for large object allocations.

Heap usage

Use the graph of heap usage to identify trends in application memory usage. If the memory footprint is larger than expected, a heap analysis tool can identify areas of excessive memory usage. If the used heap is increasing over time, the application might be leaking memory. A memory leak happens when Java applications hold references to objects that are no longer required. Because these objects are still referenced, they cannot be garbage collected and contribute to memory requirements. As the memory consumption grows, more processor resources are required for garbage collection, leaving fewer for application work. Eventually the memory requirements can fill the heap, leading to an `OutOfMemoryError` exception, and an application failure.

When monitoring a WebSphere Real Time for Linux JVM, you see a used heap graph that has a typical pattern of regular spaced collections, like the following screen capture.



Pause times

Use the graph of pause times to assess the performance affect of garbage collection. When garbage collection is running, all application threads are paused. For some applications, such as batch-processing, long pauses are not a problem. For other applications, such as GUI applications or applications that interact with other systems, long garbage collection pauses might not be acceptable.

Longer garbage collection pauses are often associated with *better* application throughput and are not a performance problem. Spending extra time in garbage collection can often lead to improved memory allocation and memory access times. The aim of garbage collection tuning is to have reduced pause times only if low response times are required.

Summary table

The summary table shows garbage collection metrics, including mean pause time, mean interval between garbage collections, and the amount of time spent in

garbage collection. The time an application spends in garbage collection must not be taken as a performance metric itself. Some garbage collection policies, such as the generational concurrent (*gencon*) policy, can take more time in garbage collection but still provide improved application performance.

Tuning recommendations

The Health Center provides general tuning recommendations and advice. In exceptional cases, further fine-tuning might be required. The Health Center does not know what your quality of service requirements are, therefore the recommendations are not always useful. For example, a suggested change might improve application efficiency but increase pause times, which might not be best for your application. The tuning recommendations also indicate if the application seems to be leaking memory. However, the Health Center cannot distinguish between naturally increasing memory requirements and memory that is being held when it is no longer required.

Object allocations

Use the object allocations view to identify which code is allocating large objects. You can use low and high-threshold values to specify the object range that triggers collection of the call stack information.

The view displays a table showing the following information:

- The size of the allocated object.
- The time of allocation.
- The code location of the allocation request.

Select a row in the table to display the call stack contents at the time of the selected allocation request.

Controlling the collection of object allocation data

When the Health Center is connected to a live agent, the following controls are available in the object allocation view:

A check box to enable the collection of object allocation call stacks
By default, collection is not enabled.

Stack trace depth
This control limits the collection of data to the specified number of stack entries. By default, five stack entries are collected.

Low threshold value
Data is collected for allocations of objects that are larger than the value specified.

High threshold value
Data is collected for allocations of objects that are smaller than the value specified.

You can specify the threshold values using bytes, kilobytes, or megabytes. The precise format of a value is `nnnn[k|m]`, where `nnnn` is the numeric value, `k` is an optional indicator for kilobyte, and `m` is an optional indicator for megabyte. For example:

- 4096 is the value 4096 bytes.
- 830k is the value 830 kilobytes.

- 2m is the value 2 megabytes.

Viewing data for a particular time period

You can select the time interval for displaying data, and making recommendations, by using cropping. For further information about cropping, see "Cropping data" on page 325.

Related concepts

"Cropping data" on page 325
You can change the time period for which data is displayed and on which recommendations are based.

Garbage collection references

Links to some websites for more information about garbage collection.

You can analyze and understand garbage collection diagnostic output through the following links:

- *Garbage collection policies, Part 1* explains the different garbage collection policies and their characteristics. Part of the *Java technology, IBM style* series.
- *Garbage collection policies, Part 2* explains what to consider when choosing a garbage collection policy, and how to get guidance on your choice from the verbose garbage collection logs. It describes the kind of information that is available from verbose garbage collection logs and presents two case studies. Part of the *Java technology, IBM style* series.
- *Fine-tuning Java garbage collection performance* tells you how to detect and troubleshoot garbage collection problems with the IBM implementation of the Java virtual machine.
- *Java diagnostics, IBM style, Part 2: Garbage collection with the IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer* discusses the garbage collector and memory visualizer, including some tutorials and example scenarios.
- *IBM Systems Journal: Tuning Garbage Collection with IBM Technology for Java* discusses tuning garbage collection for IBM i. Many of the principles are generally applicable.
- *The developerWorks® Java zone* provides all Java content for you to browse.
- *Java Diagnostics Guide; Memory Management* provides more details about garbage collection and instructions about adjusting garbage collection parameters.

I/O perspective

This perspective provides information about I/O activities performed by the target Java Virtual Machine (JVM).

Applications monitored by the Health Center might perform input or output (I/O) tasks as they run. The I/O perspective gives you information about these activities. You can use this perspective to help you solve problems such as when the application fails to close files.

The I/O perspective provides information about three aspects.

- File open events
- File close events
- Details of files that are currently open

The information is presented in one of three views.

File open view

This view reports the number of files currently held open by the target application. Use this view to find out if the number of open files is increasing. An increasing number indicates that the application might not be closing file handles after use.

File I/O view

This view shows information about each file open or file close event. Use this view to help you identify problems with I/O "bottlenecks".

Open file details view

This view shows information about the files currently held open by the target application. The information includes the file name, and the time it was opened. You can filter the information in this view by using the text box above the view. For more information about filtering, see "Filtering" on page 326.

Locking perspective

Review lock usage and identify possible points of contention.

Multi-threaded applications need to synchronize, or lock, shared resources to keep the state of the resource consistent. This consistency ensures that the status of one thread is not changed while another thread is reading it.

When locks are used in high-load applications that are deployed on systems with a large number of processors, the locking operation can prevent the application from using all the available processing resources.

The **Locking perspective** profiles lock usage and helps identify points of contention in the application or Java Runtime that prevent the application from scaling.

Using the Locking perspective

The **Locking perspective** provides information in graph and table form that helps you understand any contention caused by locking.

Information is shown for two kinds of locks:

Java monitors
: synchronized Objects in the Java application, provided as part of the Java Class Libraries, middleware, independent software packages, or application code.

System monitors
: locks that are part of the Java Runtime itself.

Java monitors are shown by default and are most useful in resolving application contention issues. To show the system monitors, use the filter icon in the top right of the table or plot.

Garbage collection time is removed from hold times for all monitors held across a garbage collection cycle.

Understanding the bar chart

The bar chart gives an overview of how contended the application locks are.

The height of the bars represents the slow lock count and is relative to all the columns in the graph. A slow count occurs when the requested monitor is already owned by another thread and the requesting thread is blocked.

The color of each bar is based on the value of the % miss column in the table. The gradient moves from red (100%), through yellow (50%), to green (0%). A red bar indicates that the thread blocks every time that the monitor is requested. A green bar indicates a thread that never blocks.

Only the most contended monitors are shown.

Understanding the table

The Monitors table shows the data for each monitor listed:

Table 18. Monitors table

Column heading	Description
% miss	The percentage of the total Gets, or acquires, for which the thread trying to enter the lock on the synchronized code had to block until it could take the lock.
Gets:	The total number of times the lock has been taken while it was inflated.
Slow:	The total number of non-recursive lock acquires for which the requesting thread had to wait for the lock because it was already owned by another thread.
Recursive:	The total number of recursive acquires. A recursive acquire occurs when the requesting thread already owns the monitor.
% util:	The amount of time the lock was held, divided by the amount of time the output was taken over.
Average hold time:	The average amount of time the lock was held, or owned, by a thread. For example, the amount of time spent in the synchronized block, measured in processor clock ticks.
Name:	The monitor name. This column is blank if the name is not known.

The table lists every monitor that was ever inflated. The % miss column is of initial interest. A high % miss shows that frequent contention occurs on the synchronized resource protected by the lock. This contention might be preventing the Java application from scaling further.

If a lock has a high % miss value, look at the average hold time and % util. If % util and average hold time are both high, you might need to reduce the amount of work done while the lock is held. If % util is high but the average hold time is low, you might need to make the resource protected by the lock more granular to separate the lock into multiple locks.

Understanding lock names

The monitor names include an object address, shown in square brackets, and the type of the lock. For example, when synchronizing on an object with class Object, the monitor name includes an address and `java/lang/Object`.

Locking on AIX

AIX architecture means that locking works differently from other platforms. On AIX, more locks might be shown as badly performing, especially system monitor locks. This is expected behavior on AIX.

Resolving lock contention

Performance can be improved using different approaches for dealing with locks.

There are two mechanisms for reducing the rate of lock contention:

- Reducing the time during which the lock is owned when taken. For example, limiting the amount of work done under the lock or in the synchronized block of code.
- Reducing the scope of the lock. For example, using a separate lock for each row in a table instead of a single lock for the whole table.

Reducing the hold time for a lock

A thread must spend as little time holding a lock as possible. The longer a lock is held, the more likely it is that another thread tries to obtain the lock. Reducing the duration that a lock is held reduces the contention on the lock and enables the application to scale further.

When a lock has a long average hold time, examine the source code to see if these conditions apply:

- All the code run while the lock is held is acting on the shared resource. Move any code in a lock that does not act on the shared resource outside the lock so that it can run in parallel with other threads.
- Any code run while the lock is held results in a blocking operation; for example, a connection to another process. Release the lock before any blocking operation is started.

Reducing the scope of a lock

The locking architecture in an application must be granular enough that the level of lock contention is low. The greater the amount of shared resource that is protected by an individual lock, the more likely it is that multiple threads will try to access the resource at the same time. Reducing the scope of the resource protected by a lock reduces the level of lock contention and enables the application to scale further.

Locking references

Links to some Web sites for more information about locking issues.

The following resources might help you to understand Java locking issues:

- *How the JIT compiler optimizes code:* describes inlining.
- *Synchronization optimizations in Mustang* explains how escape analysis can affect synchronization.

- *The Java Lock Monitor* explains the data used by the locking perspective is identical to that provided by the Java Lock Monitor.
- *Java diagnostics, IBM style, Part 3: Diagnosing synchronization and locking problems with the Lock Analyzer for Java* provides more details and case studies on resolving locking issues.

Native memory perspective

The native memory perspective provides information about the native memory usage of the process and system being monitored.

Note: This version of Health Center does not provide a native memory perspective view for the z/OS 31-bit or z/OS 64-bit platforms.
Native memory is the memory provided to the Java process by the operating system. The memory is used for heap storage and other purposes. The native memory information available in the native memory perspective view varies by platform but typically includes the following:

Table 19. Memory information values

Name	Description
Free Physical Memory	The amount of physical memory (RAM) free on the monitored system.
Process Physical Memory	The amount of physical memory (RAM) currently in use by the monitored process. On some platforms, this memory is called "resident storage" or the "working set".
Process Private Memory	The amount of memory used exclusively by the monitored process. This memory is not shared with other processes on the system.
Process Virtual Memory	Total process address space used.

More detailed discussions on understanding native memory usage can be found in two developerWorks articles:http://www.ibm.com/developerworks/java/library/j-nativememory-linux/ and http://www.ibm.com/developerworks/java/library/j-nativememory-aix/.

The perspective provides two views.

Native memory table view
This view displays a table containing the latest, minimum, and maximum values for all the available native memory information. You can use this table to see the most recent memory usage information for your monitored application.

Native memory usage view
This view plots the process virtual memory and process physical memory on a graph. The information presented helps you to monitor the native memory usage by processes. By comparing this graph to the Used Heap graph in the garbage collection perspective, you can see whether the amount of memory used by an application is due to the size of the Java Heap or due to the memory allocated natively.

Profiling perspective

Understanding the work performed by a Java application helps you to tune performance and to diagnose functional issues.

The Profiling perspective shows you which methods are run most often, and in which order.

Method profiling

You can use method profiling to see the methods that consume the most resources.

The profiling perspective shows method profiles and call hierarchies. The profiler takes regular samples to see which methods are running. Only methods that are called often, or take a long time to complete, are shown.

Reducing the resource usage when collecting method profiling data

In general, the profiling provided by the Health Center has little effect on the performance of monitored applications. When monitoring applications with deep stack traces, the use of computer resources might be more significant.

When the reduced overhead mode is enabled, the tree columns contain zeros. The Invocation Paths and Called Methods views are unavailable. The self columns continue to update. To enable the tree columns and disabled views, restart the monitored Java Virtual Machine (JVM) and reconnect the Health Center.

See “Monitored application runs out of native memory or crashes” on page 322 for more information.

Inlining

Within the Health Center, collections of methods are organized into structures called trees. Inlining is the process by which the trees of smaller methods are merged into the trees of their callers. Inlining speeds up method calls that are run frequently. The compiler might even inline methods that are not marked final. Inlined methods do not register on the method profile after they are inlined. A method might briefly show as hot before dropping to the bottom of the method profile table. The result is that time spent in the calling method suddenly increases.

Statistical profiling

The profiler is a statistical profiler, sampling the call stacks periodically rather than recording every method that is run. Methods that do not run often, or methods that run quickly, might not show in the profile list. Methods compiled by the Just-In-Time (JIT) compiler are profiled, but methods that have been inlined are not.

Performance tuning

Optimizing the code only produces a significant effect if most of the time is being spent running application code. If time is being spent on I/O, on locks, or in garbage collection, direct your performance tuning efforts to these areas instead. The Health Center draws attention to problematic garbage collection or locking.

Method Profile view:

The Method Profile table shows which methods are using the most processing resource.

Methods with a higher Self (%) value are described as “hot”, and are good candidates for optimization. Small improvements to the efficiency of these methods

might have a large effect on performance. Methods near the bottom of the table are poor candidates for optimization. Even large improvements to their efficiency are unlikely to affect performance, because they do not use as much processing resource.

Column Heading	Description
Self (%)	The percentage of samples taken while a particular method was being run at the top of the stack. This value is a good indicator of how expensive a method is in terms of using processing resource.
Self	A graphical representation of the Self (%) column. Wider, redder bars indicate hotter methods.
Tree (%)	The percentage of samples taken while a particular method was anywhere in the call stack. This value shows the percentage of time that this method, and methods it called (descendants), were being processed. This value gives a good guide to the areas of your application where most processing time is spent.
Tree	A graphical representation of the Tree (%) column. Wider, redder bars indicate hotter method stacks.
Samples	The number of samples taken while a particular method was being run at the top of the stack.
Method	A fully qualified representation of the method, including package name, class name, method name, arguments, and return type.

You can optimize methods by reducing the amount of work that they do or by reducing the number of times that they are called. Highlighting a method in the table populates the call hierarchy views.

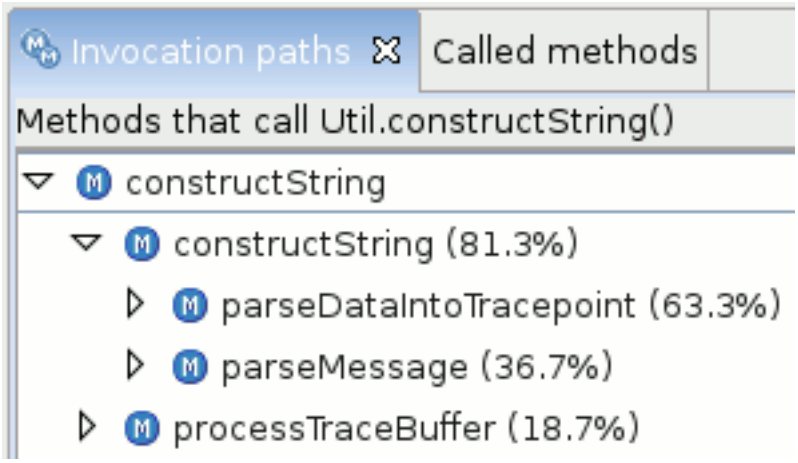
Filter the contents of the method profile table using the text box above the table. See the filtering help topic for more information.

Additionally, when you select the **Hide low sample entries**, the table does not list any entries that have a sample count of less than 2. Use this option if your table contains many entries that are not obvious candidates for optimization to improve the performance of the table.

Invocation Paths:

The **Invocations paths** tab shows the methods that called the highlighted method.

If more than one method calls the highlighted method, a weight is shown in parentheses. For any method, the sum of the percentages of its calling methods is 100%. The following example shows that a method Util.constructString() is often called by another constructString() method (81.3% of samples). The Util.constructString() method is also called occasionally by processTraceBuffer() (18.7% of samples). The top level constructString() node has two children.



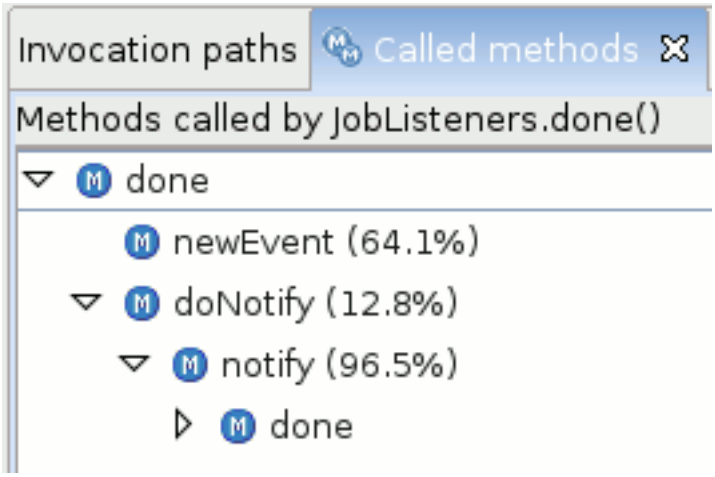
In this case, you have two strategies for optimization. The first is to make the Util.constructString() method more efficient. The second is to reduce how often it is called. Reducing how often processTraceBuffer() calls constructString() makes less difference than halving how often constructString() calls Util.constructString().

Called methods:

The **Called methods** tab shows the methods that were called by the highlighted method. In other words, they show where the highlighted method is doing its work.

If only the highlighted method is shown, no methods called by that method were sampled. Either the methods called ran quickly, or they were inlined. If the method has children in the tree, the percentages typically do not add up to 100%. The percentages for child methods never add up to more than 100%. The difference in percentages indicates the time spent in the body of the highlighted method.

In the following example, the method JobListeners.done() calls two methods, newEvent() and doNotify(). For 64.1% of the time that JobListeners.done() was on the stack, newEvent() was also on the stack. For 12.8% of the time that JobListeners.done() was on the stack, doNotify() was also on the stack. Therefore, 23.1% (that is 100% -64.1% -12.8%) of the time was spent in JobListeners.done() itself.



Note: Percentages refer only to the immediate parent node, hence for 96.5% of the time that doNotify() was on the stack, notify() was also on the stack.

The **Called methods** tab is less useful for performance tuning than the **Invocation paths** tab. Time spent processing children is not counted as time spent processing the parent. A lightweight method calling some inefficient children is not placed high in the method profile table. Any inefficient child methods typically show up in the method profile table anyway.

Timeline:

The **timeline** tab shows when the methods were invoked.

The method profiling timeline gives a visual indication of when a method was invoked in your application. You can use the graph to see if a method is used regularly throughout the lifecycle of your application. Some methods might be used only at a specific stage in the lifecycle, such as startup. This information can help you decide if the method is a good target for optimization.

Method profiling references

Links to some Web sites for more information about method profiling.

The following resource might help you to understand how to analyze method profiles:

- *How the JIT compiler optimizes code* is a section from the Java Diagnostics Guide that covers inlining.

WebSphere Real Time perspective

Unusual or exceptional aspects of application performance might be indicated by "outliers". This perspective helps you identify and analyze trace information about events that might appear inconsistent with expected application behavior.

Health Center gives you the tools required to:

- Identify outlier events.
- Filter trace information to highlight outlier events.
- Present outlier information in a timeline or histogram view.

The WebSphere Real Time perspective within Health Center enables you to answer questions about application performance, such as:

- During an application run, how often did a specific operation complete on schedule?
- Did any instance of the operation take a different amount of time to complete, in comparison to other instances?
- What are the maximum, minimum, and mean times required to complete an operation?
- What is the key factor - the "determinism factor" - affecting the performance of the operation?

The WebSphere Real Time perspective is supported when Health Center connects to applications running on one of the following platforms:

- WebSphere Real Time for Real Time Linux version 2.0 SR 3, or newer.
- WebSphere Real Time version 2.0 SR 2 with APAR IZ61672, or newer.

When Health Center detects that you are connecting to a WebSphere Real Time application, the perspective is enabled automatically. If you attempt to use the perspective while not connected to a WebSphere Real Time application, a warning message is displayed in the status bar, or the appropriate view window.

If no data is available for a view, a message reports the problem in the window. Common causes of data not being available include:

- The selected trace event is not enabled on the target application.
- No target events occurred while the trace was running.

Introduction to the WebSphere Real Time perspective

The WebSphere Real Time perspective (WRTP) helps you trace specific WebSphere Real Time (WRT) events over time.

The perspective helps you identify unusual or exceptional events that might occur when you run a WRT application. The trace information can be presented in various ways, including linear or logarithmic scales, and histograms. WRTP provides some pre-defined trace point views that are especially helpful.

Example traces include:

- Data about class loading, which lets you identify factors that have a significant impact on application performance.
- Java method execution.

Each view in WRTP represents a specific JVM or application operation. A view includes the following information:

- The component to which the specific target operation belongs.
- An entry trace point, representing the start of a specific target operation and a parameter within that operation.
- One or more exit trace points, representing the end of a specific target operation and a parameter within that operation.
- One or more information trace points, enabling you to filter specific detail from among the data collected during the trace.

The predefined trace point views are supplied as a resource bundle, and are automatically provided within the Health Center GUI. You can create and customize more views, and make them available to Health Center by adding them to a custom view store.

Data about trace points is recorded to help with the analysis. For example, each time the application reaches an entry trace point, the operation start time is recorded. Similarly, when the corresponding exit point is reached, the time is recorded and the total time to perform the operation is calculated. This data is used for graphical displays of information, and also for determinism calculations.

Within the WRTP, you can choose from predefined or customized views. Predefined views include:

- Class loading, showing the time spent in class loading.
- Incremental garbage collection, showing the time taken by the global garbage collection cycle.
- JIT compilation, showing the time spent in various compilation phases.
- Synchronous garbage collection, showing the time spent in synchronous garbage collection.

- User driven garbage collection, showing the time taken in garbage collection cycles invoked by the application.

Customized views are described in "Customizing the WebSphere Real Time perspective" on page 319.

Setting preferences for the WebSphere Real Time perspective

You can control how views appear and operate within the perspective.

The behavior of the perspective is affected by values set in the preferences menu. The value also applies for subsequent tasks.

There are two categories of preferences:

- Custom view preferences
- Display preferences

Custom view preferences

This preference category has one value only. You can specify the location of a customized view definition file. For more information about view definition files, see "Customizing the WebSphere Real Time perspective" on page 319.

Display preferences

This preference category provides values that affect the default behavior of display components. You can specify whether the default Y-axis display of plot or histogram views is presented with a logarithmic scale or not.

For the histogram display, you can select the number of intervals presented. You can also choose to exclude empty intervals from the display.

For more information about views, see "Views within the WebSphere Real Time perspective."

Views within the WebSphere Real Time perspective

Each view within the WebSphere Real Time perspective presents data in specific sections of the display.

The controller window

The controller window provides the tools for you to select views of WebSphere Real Time data. There are two main tasks you can perform using the controller window.

Manage custom views
: You can create a customized view, and add it to the list of available views. For more information about creating customized views, see "Customizing the WebSphere Real Time perspective" on page 319.

Select different views
: You can select different views, using a combination box. The box is populated initially with predefined views. Customized views also appear in this box if a custom view definition file has been created and identified in the preferences.

  All predefined views are identified by a `System view:` prefix. All customized views are identified by a `Custom view:` prefix.

The outlier plot window

This window displays event data as a simple plot graph. The X-axis of the graph shows the actual time when an event took place. The Y-axis shows the time taken for the event to occur. For convenience, the Y-axis values can be adjusted to display using a logarithmic scale.

When you hover over data in the plot window, a window opens providing details of the trace point associated with the event.

The histogram window

This window provides an alternative display of data. It shows a histogram representation of the data displayed in the outlier plot window. For example, in the predefined class loading view, the histogram representation shows how many class loading events took 0 - 1 ms to complete, how many events took 1 - 2 ms to complete, and so on.

The summary window

This window displays various statistics, calculated from the data presented in the plot window. The statistics include:

- Total events processed.
- Maximum time taken.
- Minimum time taken.
- Mean value for time taken.
- Median value for time taken.
- The standard deviation.

Recommendations and analysis window

This window displays the results of analyzing the collected data. The results are in the form of a determinism score. If the number of data samples is too low, the Health Center warns you that the determinism score might not be accurate. In particular, for Java method-based views, where the view descriptor might match multiple methods, a warning is displayed reporting that multiple methods have been matched.

The determinism score is calculated as follows:

1. Select all the data points in the plot window.
2. Calculate the median data point value - for example, the median time taken for a class loading event.
3. Find how many events fall within the following ranges:
   - Median plus or minus 20% of the median value
   - Median plus or minus 40% of the median value
   - Median plus or minus 60% of the median value
   - Median plus or minus 80% of the median value
   - Median plus or minus 100% of the median value
4. Calculate the average number of events for the ranges.
5. The average number is the determinism score, expressed as a percentage.

The determinism score can be interpreted as shown in Table 20 on page 319.

Table 20. Interpreting the meaning of a determinism score

Score	Meaning
70 or less	A very poor result. There is a wide distribution of results for the event, indicating uneven performance.
70 - 80	A poor result.
80 - 90	A good result.
90 or more	A very good result. The results are distributed closely around the median value, indicating consistent performance.

Customizing the WebSphere Real Time perspective

You can create, edit, and delete custom views within the WebSphere Real Time perspective (WRTP).

Custom views can be managed using the "custom view management wizard". The views are defined in a custom view definition file. The location of this file is set in the perspective preferences. For more information about this setting, see "Setting preferences for the WebSphere Real Time perspective" on page 317.

Creating a custom view

Use the custom view management wizard to create a custom view. Invoke the wizard by clicking the **Add custom view** button displayed in the controller view. If you have not selected a custom view definition file preference, the first page of the wizard lets you select a file.

Follow the steps presented by the wizard to create a custom view.

When the wizard finishes, the view is added to WRTP and is available immediately.

A view shows data only when the required trace settings are provided for a target JVM.

Editing a custom view

To edit a custom view, select the view in the drop-down box, then click the **Edit view** button. The wizard starts in edit mode. This mode lets you modify one or more aspects of the view. You cannot modify the name of a view.

Note: The **Edit view** button is enabled only when a custom view is selected in the drop-down box.

Deleting a custom view

To delete a custom view, select the view in the drop-down box, then click the **Delete view** button. The view is deleted. Any data associated with the view is also deleted.

Note: The **Delete view** button is enabled only when a custom view is selected in the drop-down box.

Troubleshooting

Troubleshooting information for some common problems. One method of debugging involves looking in the log files, and this information explains how to do that.

Use the navigation on the left to see the common problems available in this section.

Log files

Any output produced by the Health Center client is written to the main ISA logs. You access it by selecting **Support**, then **View Log** and **Support**, then **View Trace** under the **Help** menu. Look here first if you are experiencing problems with the Health Center tool in ISA. The agent will write to a log file in your temporary directory.

Cannot connect to an application

Possible solutions if the application you want to monitor does not appear in the connection dialog list.

Before any application can be monitored, the Java Virtual Machine (JVM) it is running on must have a Health Center agent installed. See the agent installation instructions.

Is the Health Center agent installed correctly?

Check the Health Center agent installation. See “Installing the Health Center agent” on page 293 for more information.

Has the application been enabled for monitoring?

Check that the application has been enabled for monitoring. See “Starting a Java application with the Health Center agent enabled” on page 294 for more information.

Check that the agent and application are running

Check the application to see if it has been started. Check that the agent is running on the application. If the agent has started successfully, you normally see a message like `INFO: Health Center agent started on port 1972` in the application console. The port number is also written to the `healthcenter.<pid>.log` file in the users temporary directory. The `<pid>` is the process ID for the agent that is listening on that port.

Check that the application is still running. Sometimes applications end unexpectedly early.

Connection problems are also possible if the monitored Virtual Machine (VM) is running, but there are no more live application threads.

Check for suspended applications

If the monitored VM has been suspended, the connection dialog cannot connect to the monitored VM and might timeout.

Check firewalls

When the monitored application is not on the same workstation as the client, the client must be able to access the monitored application remotely. If the remote workstation is protected by a firewall, a port must be opened in the firewall to enable the Health Center agent to listen for connections. Firewalls can also cause timeouts when scanning for Health Center agents on a remote machine. In these cases, specify the exact Health Center port, and clear the **Scan next ports for available ports** option.

Check network interfaces

If the system running the monitored application has multiple network interfaces, the agent might listen on a different interface to the one the client uses. To set the interface that the agent listens on, use system properties. To use a specific network interface, run the server with the follow property:

```
-Djava.rmi.server.hostname=<preferred_ip_address>
```

The <preferred_ip_address> determines the interface used by the agent.

Check authentication

If authentication is enabled on the monitored application, ensure that security credentials have been entered on the first page of the connection wizard. Without these credentials, the monitored application might not appear in the application list on the second page of the connection wizard.

Check that application threads are running

The Health Center agent shuts down when it detects that all application threads have terminated. In some cases, you do not want the Health Center to shut down. For example, an application which exports objects to an external RMI registry stays alive to allow RMI connections, but there are no active application threads. The Health Center agent cannot find application threads, so it terminates. To ensure that the Health Center agent keeps running, add the **keepAlive** option to the Health Center launch parameters:

```
-agentlib:healthcenter=keepAlive
```

Note: A side-effect of the **keepAlive** option is that the monitored JVM does not terminate.

Cleaning up temporary files

The Health Center client uses temporary files to hold work in progress. You can save disk space by removing these temporary files at regular intervals.

The Health Center client stores temporary data in the operating system temporary directory. Filenames have the format `healthcenter`*`[xxxx]`*`Source`*`[xxxx]`*`.tmp`. Remove these temporary data files regularly to avoid excessive use of disk space. The files are not automatically removed because they can be used by the Health Center for analysis tasks in offline mode.

For Java Virtual Machines at Java 5 SR9 and earlier, or Java 6 SR4 and earlier, the Health Center agent also stores data to disk. Delete the `perfmon.out` file occasionally if you typically start the agent from the command line:

```
-agentlib:healthcenter -Xtrace:output=perfmon.out
```

Data disappears

If the Health Center detects that it is going to run out of memory, it automatically removes some older stored data.

Health Center runs a data truncation job at regular intervals to help prevent problems that occur when running out of memory. Each time the job runs, Health Center checks to see if it might run out of memory. If required, the Health Center removes the oldest half of the stored data, based on the time the data was generated.

By default, the data truncation job runs every 30 seconds. To change the time interval, modify the value in the Data Storage section of the Health Center preferences.

GUI unresponsive

The GUI might seem unresponsive due to the refresh rate of the Health Center.

The Health Center refreshes every ten seconds. This delay can sometimes make the GUI seem unresponsive.

Hangs

Information about the environment variable **DBUS_SESSION_BUS_ADDRESS** on Linux which can cause the Health Center to hang.

On some versions of Linux, the Health Center can go into an endless loop when trying to open a file and seem to stop. If you log on to root from a normal user account, use `su -` instead of `su`, otherwise you will inherit the **DBUS_SESSION_BUS_ADDRESS** environment variable from your normal user account. This variable is known to cause problems.

Monitored application runs out of native memory or crashes

Information about running the Health Center in a lower computer resource usage mode.

The Health Center provides a mode for lower computer resource usage which you enable by adding the option **level=low** to the Health Center command line. Alternatively, the `healthcenter.properties` file, as found in `$JAVA_HOME/lib`, can be edited and the com.ibm.java.diagnostics.healthcenter.data.collection.level property changed from **full** to **low**.

On systems with many processors, and where the monitored application has deep stack traces, the Health Center agent can sometimes consume unacceptable amounts of native memory. On certain Virtual Machine (VM) levels, this consumption could cause a failure in the VM. Configuring the Health Center for lower resource usage might help prevent problems.

For more information about reducing resource usage during data collection, see "Controlling the amount of data generated" on page 300.

No data present

There are several questions that you must consider when determining why no data is present. The Health Center is most likely not to show updated data because your connection to the agent is not functioning correctly or your application is not doing enough work.

Checking for a successful connection

If the Health Center successfully connects to an application, the message `Connected to <host>:<port>` displays in the bottom left status line. If no connection is made, `Unable to connect to the live application` is displayed.

If you cannot connect, check that your application was launched with the correct arguments for your Java version. See "Platform requirements" on page 291 for further information about using the Health Center with different versions of Java.

Check that you connected the client using the connection wizard. A message dialog tells you when a successful connection is made.

Check that your firewall allows you to connect to the ports.

Is your application doing anything?

Data collected by the agent is buffered before being transferred to the client for processing. If your application spends much time not running methods, for example when waiting for GUI input, or does not trigger regular garbage collections, the Health Center client data might take some time to display and update.

Has the application been running for some time?

When connecting for the first time to a long-running application, there might be a delay before data is displayed. The delay is a known limitation.

Are any trace options set?

The Health Center is not compatible with the trace option **-Xtrace:none**. If this option is set, no garbage collection or profiling data is available.

Is the Just-In-Time (JIT) compiler on?

Profiling data is not available if the JIT compiler on the profiled application is disabled.

Are you using the Java Debug Wire Protocol (JDWP)?

Profiling data is not available if you are debugging using JDWP on the profiled application.

No I/O information present

You might not see any I/O information when using Java 6 on the Windows platform.

If you have the latest agent installed, you can obtain I/O information by adding `-Xtrace:maximal=io` to the command line of the application you are monitoring.

No method names showing

When connecting again to an agent, you might not see expected method names.

For Java 5 SR 9 and earlier or Java 6 SR 4 and earlier, if a previous connection was made to the agent, you can view method names only for classes loaded since the previous client was disconnected.

Only the first character of file names showing

When viewing file names, you might see only the first character.

The shortening of file names in this way is a known problem on Windows when using JVMs earlier than Java 6 SR 8.

Out of memory errors and ISA 4.1

When using IBM Support Assistant (ISA) 4.1, the Health Center might run out of memory while processing large files.

Processing large files using the Health Center might fail sometimes with the `java.lang.OutOfMemoryError` message. This can be due to an insufficient Java heap size. By default, IBM Support Assistant 4.1 has a maximum Java heap size of 256 MB. You should run the Health Center with a heap size of at least 512 MB.

To set the maximum Java heap size for ISA, add a property value to the `rcpinstall.properties` file in the ISA workspace. Add or update the value for the **vmarg.Xmx** property. For example, to set a maximum heap size of 512 MB, add the line:

```
vmarg.Xmx=-Xmx512m
```

You must restart ISA for the changes to take effect.

On Windows, you normally find the `rcpinstall.properties` file in:

```
<home drive>\<home path>\IBM\ISAv41\.config\rcpinstall.properties
```

for example:

```
C:\Documents and Settings\Administrator\IBM\ISAv41\.config\rcpinstall.properties
```

On Linux, you normally find the `rcpinstall.properties` file in:

```
<home>/ibm/isa41/.config/rcpinstall.properties
```

Printing

Printing is not supported in the Health Center.

This release of the Health Center does not support printing of information or reports.

Showing the Status perspective

Normally, the system status summary is always visible. To see the full system status, use the **Status** perspective.

All perspectives show a summary of the system status. Use the **Status** perspective to see the full system status. To open the **Status** perspective, click the toolbar icon:


.

Problems when using WebSphere Application Server - Community Edition

There is a conflict between the Health Center agent and the MBeanServer used by WebSphere Application Server - Community Edition.

Resolve the problem by changing the behavior of the Heath Center agent. Add the following property to the command line when launching the application you want to monitor:

```
-Dcom.ibm.java.diagnostics.heathcenter.use.platformmbeanserver=true
```

When using this property, the Heath Center agent attempts to use the MBeanServer that is created by the running application. You might also need to delay the start of the Health Center agent to ensure that the application has started the MBeanServer. Include the following property to introduce a short delay in the Health Center agent startup.

```
-Dcom.ibm.java.diagnostics.healthcenter.agent.start.delay.seconds=<delay>
```

where **<delay>** is the number of seconds that the Health Center agent pauses before starting.

Resetting displayed data

To assess the performance affect or attributes of a particular function in the program that you are monitoring, use the Health Center to remove all currently analyzed data from the views.

The Health Center provides information about a monitored application. To concentrate on specific details, you can isolate some data. For example, if you want to assess the most active method during a file load operation, you can isolate the data recorded for program actions that take place when you use the application GUI to load a file

Resetting the data

Health Center provides this ability through a menu option **Data** and **Reset data**, duplicated on the toolbar. **Reset data** immediately deletes the data stored in the data model in all views. You see data collected only after the time that you start the **Reset data** function.

Limitations

- Incoming data is ignored after a data reset based on the timestamp at the GUI of the client. If the system time on the agent machine is not the same as the system time on the client machine, **Reset data** does not behave as expected.

Cropping data

You can change the time period for which data is displayed and on which recommendations are based.

What is cropping?

Cropping involves selecting a subset of data by specifying a time interval on a graph. The time interval is used to limit the data displayed by the Health Center, and also affects how much data is used to make recommendations. Any data recorded outside the time interval is ignored after cropping. Cropped data is not displayed in graphs and is not used for recommendations.

What can you crop?

You can crop data on graphs. Graphs are displayed in several perspectives, such as I/O and memory. Similarly, you can crop data on the time line graph within the profiling perspective.

Some data, for example environment properties, are not time-based. If data are not displayed in graph form, they cannot be cropped.

Why it is useful to crop data

If you know that a problem took place at a particular time, you might want to crop the data to concentrate on the time interval of interest. Cropping helps reduce the quantity of data to process.

How to crop data

To crop data on a graph, start by specifying the beginning of the time interval. Specify the beginning by clicking one point in the graph. Next, drag to a second point in the graph. The second point corresponds to the end of the time interval. The graph adjusts so that only the selected time interval is displayed. Data recorded outside the time interval is ignored.

How to reset cropped data

You can reset the graph so that data is no longer cropped in two ways:

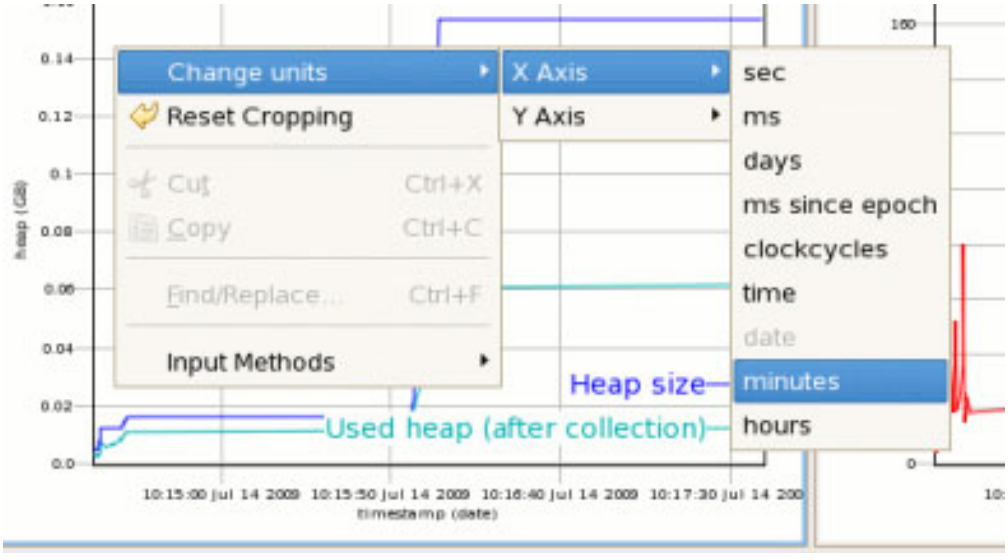
- Right-click on the graph. Select **Reset Cropping**.
- Double-click anywhere on the graph.

Controlling the units

The units displayed by the Health Center can be modified in graphs, tables and recommendations.

You can change the units of data that the Health Center displays. For example, it is possible to use calendar dates instead of relative time, or to use GB instead of MB. Unit changes are global, therefore changing the units in a graph will also adjust the units in recommendations and in tables.

To change the units, hover over a graph to open the pop-up window. Click **Change Units**, select the axis you want to change and then select the units you want Health Center to display. If you want to use absolute times instead of relative times, click **date** on the x-axis.



Filtering

Use regular expressions to filter the information displayed in views.

You can filter the output of tables, such as the method profile and classes tables, by entering expressions in the text box above the corresponding table. The filter text box accepts well-formed regular expressions. When you enter part of the name, only lines with matching content appear in the table. You can enter ^ to match the beginning of some text, such as a class name. Similarly, using $ forces a match at the end of the text.

For example, to see only packages beginning with "java", enter `^java` in the text box.

To see only method names containing ".init", enter `\.init` in the text box. The "\" is important to escape the "`.`" which otherwise matches any character.

Filter examples

The following table shows some sample filter expressions:

Filter expression	Required results
`lang`	Any line containing `lang`
`^com.ibm`	Any line beginning with `com.ibm`
`\.get`	Any line containing `.get`

Performance hints

The Health Center agent has little effect on performance. You can improve the performance of the Health Center agent in several ways.

Monitored Application: Reducing the amount of data collected

You can further minimize the agent resource usage by reducing the amount of data collected. The Health Center provides a low resource usage mode which can be enabled by adding the option `level=low` to the Health Center command line. For example:

```
-agentlib:healthcenter=level=low -Xtrace:output=perfmon.out
```

or

```
-Xhealthcenter:level=low
```

Health Center Client: Reducing the amount of data collected

Collecting less data also reduces the memory footprint of the Health Center client. For related information about collecting less data, see "Controlling the amount of data generated" on page 300.

Health Center Client: Reducing the amount of data displayed

The Health Center stores a configurable amount of historical data. Storing less historical data reduces the memory footprint of the Health Center and improves performance. To configure the age at which data is discarded and how often the Health Center deletes old data, modify the data storage settings as described in "Saving data" on page 301.

Chapter 15. Reference

This set of topics lists the options and class libraries that can be used with WebSphere Real Time for RT Linux

Options

The launcher has a set of standard options that are supported on the current runtime environment and will be supported in future releases. In addition, there is a set of nonstandard options. The default options have been chosen for best general use.

Specifying Java options and system properties

There are three ways to specify Java properties and system properties.

About this task

You can specify Java options and system properties in these ways. In order of precedence, they are:

1. By specifying the option or property on the command line. For example:
   ```
   java -Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump MyJavaClass
   ```
2. By creating a file that contains the options, and specifying it on the command line using the **-Xoptionsfile=**<*filename*> option.
3. By creating an environment variable called **IBM_JAVA_OPTIONS** containing the options. For example:
   ```
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump"
   ```

Rightmost options on the command line have precedence over leftmost options; for example, if you specify the options **-Xint -Xjit myClass**, **-Xjit** takes precedence.

Real-time options

The definition of the **-Xrealtime** option used in WebSphere Real Time for RT Linux.

The **-X** following option are applicable in the WebSphere Real Time for RT Linux environment.

-Xrealtime

Starts the real-time mode. It is required if you want to run the Metronome Garbage Collector and use the Real-Time Specification for Java (RTSJ) services. If you do not specify this option, the JVM starts in non-real-time mode equivalent to IBM SDK and Runtime Environment for Linux Platforms, Java 2 Technology, version 6.0.

The **-Xrealtime** option is interchangeable with **-Xgcpolicy:metronome**. You can specify either one to get real-time mode.

Ahead-of-time options

The definitions for the ahead-of-time options.

Purpose

No option specified:
Runs with the interpreter and dynamically compiled code. If AOT code is discovered, it is not used. Instead, it is dynamically compiled as required. This is particularly useful for non-real time and some real-time applications. This option provides optimal performance and throughput but can suffer non-deterministic delays at runtime when compilation occurs.

-Xjit: This option is the same as default.

-Xint: Runs the interpreter only, ignores code written for AOT that might be found in a precompiled jar file, and does not run the dynamic compiler. This mode is not often required, other than for debugging problems that you suspect are related to compilation or for very short batch applications that do not derive benefit from compilation.

-Xnojit:
Runs the interpreter and uses code written for AOT if it is found in a precompiled jar file. It does not run the dynamic compiler. This mode works well for some real-time applications where you want to ensure that no non-deterministic delays occur at runtime because of compilation. Code written for AOT can only be used when running with the **-Xrealtime** option. It is not supported when running in a standard JVM, that is, **-Xrealtime** is not specified.

Example

```
java -Xrealtime -Xnojit outputtest.jar.
```

Related concepts

Chapter 4, “Using compiled code with WebSphere Real Time for RT Linux,” on page 23
IBM WebSphere Real Time for RT Linux supports several models of code compilation, providing varying levels of code performance and determinism.

“The ahead-of-time compiler” on page 25
Ahead-of-Time (AOT) compilation allows you to compile Java classes and methods before you run your code. AOT compilation avoids the unpredictable timing effect the JIT compiler can have on sensitive performance paths. To ensure your code is compiled before it executes and for the highest level of deterministic performance, you can precompile your code into a shared class cache using the AOT compiler.

Related tasks

“Storing precompiled jar files into a shared class cache” on page 40
You can store all, some, or include Java classes provided by IBM into a shared class cache. This process uses the **-Xrealtime** option with javac and the admincache tool to store the classes into a shared class cache.

“Precompiling all classes and methods in an application” on page 41
This procedure precompiles all the classes in an application. It stores a set of jar files into a shared class cache. All methods in all classes in those jar files are stored into the cache. The optimized jar files have all methods compiled.

“Precompiling frequently used methods” on page 42
You can use profile-directed AOT compilation to precompile only the methods that are frequently used by the application. AOT compilation stores a set of jar files into a shared class cache using an option file generated by running the application with a special option **-Xjit:verbose={precompile},vlog=optFile**. Only the methods listed in the option file are precompiled.

“Precompiling files provided by IBM” on page 43
You can precompile files provided by IBM, for example `rt.jar`, to achieve a compromise between performance and predictability.

“Using the AOT compiler” on page 27
Use these steps to precompile your Java code. This procedure describes the use of the **-Xrealtime** option in a javac command, the admincache tool, and the **-Xrealtime** and **-Xnojit** options with the java command.

“Running the sample application using AOT” on page 80
This procedure runs a standard Java application in a real-time environment using the ahead-of-time (AOT) compiler, without the need to rewrite code. Use this sample to compare running the same application using the JIT compiler.

jxeinajar utility

The jxeinajar utility is used in WebSphere Real Time for RT Linux before SR1 to precompile Java programs for use in WebSphere Real Time for RT Linux.

Purpose

By default, jxeinajar searches for files with a .jar or .zip extension in the current working directory and all its subdirectories.

Note: If you are using WebSphere Real Time for RT Linux SR1 or above, you must migrate because the jxeinajar tool is no longer supported. See “WebSphere Real Time V1 Migration” on page 38 for information on how to do this.

You can change the input directory by specifying the **-searchPath** option. The search for input files can be extended to include subdirectories of the search path

by specifying the **-recurse** option. The optimized jar files written to the **-outPath** directory tree have the same relative structure as the input jar files. You can specify individual jar files as input to jxeinajar.

jxeinajar

Parameters

-help | -?
Shows help.

-Xrealtime
Specifies that the bootclass path that is used for processing contains real-time classes. This option must be supplied on the command line. This option is ignored if it is in the options file.

-outPath
Specifies the root directory other than the current directory or a subdirectory of current directory where the JXEs are generated. The path name can be relative and this option is mandatory.

-[no]logo
Shows the copyright message. **-logo** is a default.

-[no]verify
Verifies the bytecode in the source file. **-verify** is a default.

-searchPath
Is the path where jxeinajar searches for input files with a `.jar` or `.zip` extension in the working directory. The default is the current working directory.

-[no]recurse
-recurse extends the search for files with an extension of `.jar` or `.zip` to subdirectories of the search path. **-norecurse** is the default.

-noisy
Prints out progress messages for each class in each jar.

-verbose
Prints out progress messages for each jar. **-verbose** is a default.

-quiet
Suppresses all progress messages.

-[no]precompile
Requests precompilation (or no precompilation) of all methods.

-[no]precompileMethod*xxx*
Requests precompilation (or no precompilation) of all methods.

-optFile
Specifies an options file that contains one or more jxeinajar options. An options file can contain any option except another **-optFile** option. The format of the options file is a multiline string of options

[jar file]*
Is a list of specific jar files to process. If no files are specified, all files in the searchPath are converted.

Sample

This example processes the `main.jar` and `util.zip` input files, generating the optimized files `app/aot/main.jar` and `app/lib/util.zip`. `debug.jar` is not processed because the **-recurse** flag has not been specified.

```
app/lib/main.jar
app/lib/ext/debug.jar
app/lib/util.zip

cd app
jxeinajar -Xrealtime -searchPath lib -outPath aot
```

Prepending locations to the Java bootclasspath

When precompiling jar files shipped by IBM, to ensure that the precompiled versions of the jar files are loaded instead of the originals, the precompiled jar files must be prepended to the bootclasspath. To prepend a jar file to the bootclasspath use the **-Xbootclasspath/p** <:directories and .zip or .jar files separated by a colon> option to the Java runtime.

There are also **-Xbootclasspath:** for setting the bootclasspath and **-Xbootclasspath/a:** for appending to the bootclasspath options to the Java runtime, but they are less commonly used and will not ensure that a precompiled jar file is used.

```
java -Xrealtime -Xnojit -Xbootclasspath/p:$APP_HOME/aot/core.jar
                -classpath:$APP_HOME/aot/main.jar:$APP_HOME/aot/util.jar ...
```

Where `core.jar` is an IBM-provided file. `main.jar` and `util.jar` are your application precompiled files.

Return codes from jxeinajar

0 JAR correctly processed.
<10 JIT error and warnings.
11-20 jxeinajar errors.
>100 jar2jxe errors.

Return codes ≤ 5 are considered warnings and processing continues. If multiple warnings or errors are encountered in a single run, the highest value is returned.

To precompile only methods that are used by the application:

1. Run the application with:

   ```
   java -Xjit:verbose={precompile},vlog=optFile
   ```

   to profile the application and store the list of methods to precompile in optFile.
2. Precompile your application using the `precompileOpts` files generated from the profiling run:

   ```
   jxeinajar -Xrealtime -outPath aot -optFile vlog
   ```

Specific return codes

JIT error codes:

Compilation Failure

1 Compiler ran out of memory trying to perform the compilation of the given method.

Compilation Restriction ILNodes

2 The number of internal IL instructions for this method is too large.

Compilation Restriction RecDepth
3 The stack depth for an optimization phase has exceeded this stack memory.
Compilation Restricted Method
4 Attempt to compile a JNI or abstract method.
Compilation Excessive Complexity
5 Method is too large to compile.

Note: jxeinajar error codes caused by signals have values greater than 128. jxeinajar handles only UNIX signals, such as SIGSEGV, SIGBUS, SIGILL, SIGFPE, SIGTRAP. If one of these signals is intercepted by the jxeinajar command, the return code is 160. For any other signal, the error code is equal to the signal number, + 128, for example, Ctrl-C returns 130.

Known limitation

There is currently a limitation when carrying out ahead-of-time compilation of invalid class files. If you run jxeinajar against a jar file that contains invalid class files (and compilation is requested, that is, you have not specified **-noprecompile**), the processing of the jar file might fail with a return code of 160. As a result jxeinajar terminates without processing the rest of the jar file and therefore fails to produce a new jar file.

As a workaround, you can determine which class file is causing the problem by re-running jxeinajar with the **-noisy** option to determine which class was being compiled at the point of failure. You can then remove the invalid class from the jar file, or update it with a valid class file. Running jxeinajar against the jar file should then complete successfully (assuming it does not contain any additional invalid class files).

Standard options

The definitions for the standard options.

-agentlib:<*libname*>**[=**<*options*>**]**
Loads native agent library <*libname*>; for example **-agentlib:hprof**. For more information, specify **-agentlib:jdwp=help** and **-agentlib:hprof=help** on the command line.

-agentpath:*libname***[=**<*options*>**]**
Loads native agent library by full path name.

-assert Prints help on assert-related options.

-cp or -classpath <*directories and .zip or .jar files separated by* **:**>
Sets the search path for application classes and resources. If **-classpath** and **-cp** are not used and **CLASSPATH** is not set, the user classpath is, by default, the current directory (.).

-D<property_name>=<*value*>
Sets a system property.

-help or -?
Prints a usage message.

-javaagent:<*jarpath*>**[=**<*options*>**]**
Loads Java programming language agent. For more information, see the java.lang.instrument API documentation.

-jre-restrict-search
Includes user private JREs in the version search.

-no-jre-restrict-search
Excludes user private JREs in the version search.

-showversion
Prints product version and continues.

-verbose:[*class*,*gc*,*dynload*,*sizes*,*stack*,*jni***]**
Enables verbose output.

-verbose:class
Writes an entry to stderr for each class that is loaded.

-verbose:gc
See "Using verbose:gc information" on page 51.

-verbose:dynload
Provides detailed information as each class is loaded by the JVM, including:
- The class name and package
- For class files that were in a `.jar` file, the name and directory path of the `.jar`
- Details of the size of the class and the time taken to load the class

The data is written out to stderr. An example of the output follows:

```
<Loaded java/lang/String from /myjdk/sdk/jre/lib/i386/softrealtime/jclSC160/vm.jar
<Class size 17258; ROM size 21080; debug size 0>
<Read time 27368 usec; Load time 782 usec; Translate time 927 usec>
```

Note: Classes loaded from the shared class cache do not appear in **-verbose:dynload** output. Use **-verbose:class** for information about these classes.

-verbose:sizes
Writes information to stderr describing the amount of memory used for the stacks and heaps in the JVM

-verbose:stack
Writes information to stderr describing Java and C stack usage.

-verbose:jni
Writes information to stderr describing the JNI services called by the application and JVM.

-version
Prints out version information for the non-real-time mode. When used with the -Xrealtime option, it prints out the version information for real-time mode.

-version:<*value*>
Requires the specified version to run.

-X Prints help on nonstandard options.

Nonstandard garbage collection options

These **-X** options are used with garbage collection and are nonstandard and subject to change without notice.

These options are grouped to show those that can be used with WebSphere Real Time for RT Linux, standard non-real-time mode, and with both Metronome Garbage Collector and WebSphere Real Time for RT Linuxand IBM 32-bit SDK for Linux on Intel architecture, Java 2 Technology Edition.

Metronome Garbage Collector options

The definitions of the Metronome Garbage Collector options.

-Xgc:immortalMemorySize=*size*
Specifies the size of your immortal heap area. The default is 16 MB.

-Xgc:scopedMemoryMaximumSize=*size*
Specifies the size of your scoped memory heap area. The default is 8 MB.

-Xgc:synchronousGCOnOOM | -Xgc:nosynchronousGCOnOOM
One occasion when garbage collection occurs is when the heap runs out of memory. If there is no more free space in the heap, using **-Xgc:synchronousGCOnOOM** stops your application while garbage collection removes unused objects. If free space runs out again, consider decreasing the target utilization to allow garbage collection more time to complete. Setting **-Xgc:nosynchronousGCOnOOM** implies that when heap memory is full your application stops and issues an out-of-memory message. The default is **-Xgc:synchronousGCOnOOM**.

-Xnoclassgc
Disables class garbage collection. This option switches off garbage collection of storage associated with Java classes that are no longer being used by the JVM. The default behavior is **-Xnoclassgc**.

-Xgc:targetUtilization=*N*
Sets the application utilization to N%; the Garbage Collector attempts to use at most (100-N)% of each time interval. Reasonable values are in the range of 50-80%. Applications with low allocation rates might be able to run at 90%. The default is 70%.

This example shows the maximum size of the heap memory is 30 MB. The garbage collector attempts to use 25% of each time interval because the target utilization for the application is 75%.

```
java -Xrealtime -Xmx30m -Xgc:targetUtilization=75 Test
```

-Xgc:threads=*N*
Specifies the number of GC threads to run. The default is 1.

-Xgc:verboseGCCycleTime=N
N is the time in milliseconds that the summaries should be dumped.

Note: The cycle time does not mean that the summary is dumped precisely at that time, but rather when the last GC quanta or heartbeat that passes this time criteria.

-Xmx<*size*>
Specifies the Java heap size. Unlike other garbage collection strategies, the real-time Metronome GC does not support heap expansion. There is not an initial or maximum heap size option. You can specify only the maximum heap size.

-Xthr:metronomeAlarm=os*xx*
Controls the priority that the Metronome Garbage Collector alarm thread runs at.

where *xx* is a number from 11 to 89 that specifies the priority the metronome alarm thread should run at. Care should be taken in modifying the OS priority that the alarm thread runs at. If you specify an OS priority lower than that of any realtime thread, you will experience OutOfMemory errors because the Garbage Collector ends up running at a lower priority than realtime threads allocating garbage. The default Metronome Garbage Collector alarm thread runs at an OS priority of 89.

Related concepts

Chapter 6, "Using the Metronome Garbage Collector," on page 49
Metronome Garbage Collector replaces the standard Garbage Collector in WebSphere Real Time for RT Linux.

"Introduction to the Metronome Garbage Collector" on page 49
The benefit of the Metronome Garbage Collector is that the time it takes is more predictable and garbage collection can take place at set intervals over a period of time.

"Using verbose:gc information" on page 51
You can use the **-verbose:gc** option with the **-Xgc:verboseGCCycleTime=N** option to write information to the console about Metronome Garbage Collector activity. Not all XML properties in the **-verbose:gc** output from the standard JVM are created or apply to the output of Metronome Garbage Collector.

Garbage collection options used with the non-real-time mode

These nonstandard option can be used with the non-real-time mode only. They are not supported for use with WebSphere Real Time for RT Linux.

For options that take <*size*> parameter, you should suffix the number with "k" or "K" to indicate kilobytes, "m" or "M" to indicate megabytes, or "g" or "G" to indicate gigabytes.

-Xclassgc
Enables collection of class objects at every garbage collection. By default, this option is enabled. For Real-time Java, this option is not supported. Classes cannot be collected by the Metronome Garbage Collector.

-Xcompactgc
Compact every Garbage Collector cycle. See also **-Xnocompactgc**. By default, compaction occurs only when triggered internally.

-Xdisableexcessivegc
Disables the throwing of an OutOfMemoryError if excessive time is spent in the GC. By default, this option is off.

-Xdisablestringconstantgc
Prevents strings in the string intern table from being collected. By default, this option is disabled.

-Xenableexcessivegc
If excessive time is spent in the GC, this option returns NULL for an allocate request and thus causes an OutOfMemoryError to be thrown. This action occurs only when the heap has been fully expanded and the time spent is making up at least 95%. This behavior is the default.

-Xenablestringconstantgc
Enables strings from the string intern table to be collected. By default, this option is enabled.

-Xgcpolicy:[optthruput]|[optavgpause]|[gencon]|[metronome]
Controls the behavior of the Garbage Collector.

- The *optthruput* option is the default and delivers very high throughput to applications, but at the cost of occasional pauses.
- The *optavgpause* option reduces the time that is spent in these garbage collection pauses and limits the effect of increasing heap size on the length of the garbage collection pause. Use *optavgpause* if your configuration has a very large heap.
- The *gencon* option requests the combined use of concurrent and generational GC to help minimize the time that is spent in any garbage collection pause.
- The *metronome* option uses the Metronome Garbage Collector. If **-Xrealtime** option is specified, it will set the GC policy to be Metronome

For more information, see "Specifying garbage collection policy for non Real-Time" on page 369.

-Xgcworkpackets <*number*>
Specifies the total number of work packets available in the global collector. If not specified, the collector allocates a number of packets based on the maximum heap size.

-Xnocompactgc
Disables compaction for the Garbage Collector. See also **-Xcompactgc**. By default, compaction is enabled.

-Xnopartialcompactgc
Disables incremental compaction. See also **-Xpartialcompactgc**. By default, this option is not set, all compactions are full.

-Xpartialcompactgc
Enables partial compaction. See also **-Xnopartialcompactgc**.

-Xcompactexplicitgc
Compact on every call to System.gc(). See also **-Xnocompactexplicitgc**. By default, compaction occurs only when triggered internally.

-Xnocompactexplicitgc
Disables compaction on a call to `System.gc()`. See also **-Xcompactexplicitgc**. By default, compaction is enabled on calls to `System.gc()`.

-Xlp Requests the JVM to allocate the Java heap with large pages. If large pages are not available, the JVM will not start, displaying the error message `GC: system configuration does not support option --> '-Xlp'`. The JVM uses shmget() to allocate large pages for the heap. Large pages are supported by systems running Linux kernels v2.6 or higher, or earlier kernels where large page support has been backported by the distribution. By default, large pages are not used. See the *Diagnostics Guide*.

-Xmaxe<*size*>
Sets the maximum amount by which the garbage collector expands the heap. Typically, the garbage collector expands the heap when the amount of free space falls below 30% (or the amount specified using **-Xminf**), by the amount required to restore the free space to 30%. The **-Xmaxe** option limits the expansion to the specified value; for example `-Xmaxe10M` limits the expansion to 10 MB. By default, there is no maximum expansion size.

-Xmaxf<*size*>
Specifies the maximum percentage of heap that must be free after a garbage collection. If the free space exceeds this amount, the JVM attempts

to shrink the heap. Specify the size as a decimal value in the range 0-1, for example `-Xmaxf0.5` sets the maximum free space to 50%. The default value is 0.6 (60%).

-Xmine<*size*>
Sets the minimum amount by which the Garbage Collector expands the heap. Typically, the garbage collector expands the heap by the amount required to restore the free space to 30% (or the amount specified using **-Xminf**). The **-Xmine** option sets the expansion to be at least the specified value; for example, `-Xmine50M` sets the expansion size to a minimum of 50 MB. By default, the minimum expansion size is 1 MB.

-Xminf<*size*>
Specifies the minimum percentage of heap that should be free after a garbage collection. If the free space falls below this amount, the JVM attempts to expand the heap. Specify the size as a decimal value in the range 0-1; for example, a value of **-minf0.3** requests the minimum free space to be 30% of the heap. By default, the minimum value is 0.3.

-Xmn<*size*>
Sets the initial and maximum size of the new (nursery) heap to the specified value when using -Xgcpolicy:gencon. Equivalent to setting both **-Xmns** and -Xmnx. If you set either **-Xmns** or **-Xmnx**, you cannot set **-Xmn**. If you attempt to set **-Xmn** with either **-Xmns** or **-Xmnx**, the VM will not start, returning an error. By default, **-Xmn** is selected internally according to your system's capability. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

-Xmns<*size*>
Sets the initial size of the new (nursery) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmn**.

-Xmnx<*size*>
Sets the maximum size of the new (nursery) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmn**.

-Xmo<*size*>
Sets the initial and maximum size of the old (tenured) heap to the specified value when using **-Xgcpolicy:gencon**. Equivalent to setting both **-Xmos** and **-Xmox**. If you set either **-Xmos** or **-Xmox**, you cannot set **-Xmo**. If you attempt to set **-Xmo** with either **-Xmos** or **-Xmox**, the VM will not start, returning an error. By default, **-Xmo** is selected internally according to your system's capability. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

-Xmos<*size*>
Sets the initial size of the old (tenure) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmo**.

-Xmox<*size*>
Sets the maximum size of the old (tenure) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmo**.

-Xmoi<*size*>
Sets the amount the Java heap is incremented when using **-Xgcpolicy:gencon**. If set to zero, no expansion is allowed. By default, the increment size is calculated on the expansion size, **-Xmine** and **-Xminf**.

-Xmr<*size*>
Sets the size of the Garbage Collection "remembered set" when using **-Xgcpolicy:gencon**. This is a list of objects in the old (tenured) heap that have references to objects in the new (nursery) heap. By default, this option is set to 16 kilobytes.

-Xmrx<*size*>
Sets the remembered maximum size setting.

-Xms<*size*>
Sets the initial Java heap size. You can also use **-Xmo**. The default is set internally according to your system's capability.

Other nonstandard options

These **-X** options are nonstandard and subject to change without notice.

For options that take <*size*> parameter, you should suffix the number with "k" or "K" to indicate kilobytes, "m" or "M" to indicate megabytes, or "g" or "G" to indicate gigabytes.

-Xaot[:<*suboption*>**,***suboption***,...]**
Enables the AOT compiler if **-Xshareclasses** is also present. For details of the suboptions, see the Diagnostics Guide. See also **-Xnoaot**. By default, the AOT compiler is enabled, but it is only active in conjunction with **-Xshareclasses**.

-Xargencoding
Allows you to put Unicode escape sequences in the argument list. This option is set to off by default.

-Xbootclasspath:<*directories and .zip or .jar files separated by : >*
Sets the search path for bootstrap classes and resources. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xbootclasspath/a:<*directories and .zip or .jar files separated by : >*
Appends the specified directories, .zip, or .jar files to the end of bootstrap class path. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xbootclasspath/p:<*directories and .zip or .jar files separated by : >*
Prepends the specified directories, .zip, or .jar files to the front of the bootstrap class path. Do not deploy applications that use the **-Xbootclasspath:** or **-Xbootclasspath/p:** option to override a class in the standard API, because such a deployment contravenes the Java 2 Runtime Environment binary code license. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xcheck:jni
Performs additional checks for JNI functions. You can also use **-Xrunjnichk**. By default, no checking is performed.

-Xcheck:nabounds
Performs additional checks for JNI array operations. You can also use **-Xrunjnichk**. By default, no checking is performed.

-Xcodecache<*size*>
Sets the unit size of which memory blocks are allocated to store native code of compiled Java methods. An appropriate size can be chosen for the application being run. By default, this is selected internally according to the CPU architecture and the capability of your system.

-Xconcurrentbackground <*number*>
Specifies the number of low priority background threads attached to assist the mutator threads in concurrent mark. The default is 1.

-Xconcurrentlevel <*number*>
Specifies the allocation "tax" rate. It indicates the ratio between the amount of heap allocated and the amount of heap marked. The default is 8.

-Xconmeter:<soa | loa | dynamic>
Determines which area's usage, LOA (Large Object Area) or SOA (Small Object Area), is metered and hence which allocations are taxed during concurrent mark. The allocation tax is applied to the selected area. If **-Xconmeter:dynamic** is specified, the collector dynamically determines the area to meter based on which area is exhausted first. By default, the option is set to **-Xconmeter:soa**.

-Xdbg:<*options*>
Loads debugging libraries to support the remote debugging of applications. Specifying **-Xrunjdwp** provides the same support. By default, the debugging libraries are not loaded, and the VM instance is not enabled for debug.

-Xdbginfo:<*path to symbol file*>
Loads and passes options to the debug information server. By default, the debug information server is disabled.

-Xdisablejavadump
Turns off javadump generation on errors and signals. By default, javadump generation is enabled.

-Xfuture
Turns on strict class-file format checks. Use this flag when you are developing new code because stricter checks will become the default in future releases. By default, strict format checks are disabled.

-Xint Makes the JVM use only the Interpreter, disabling the Just-In-Time (JIT) compiler. By default, the JIT compiler is enabled.

-Xiss<*size*>
Sets the initial Java thread stack size. 2 KB by default.

-Xjit[:<*suboption*>**,***suboption*,**...]**
Enables the JIT. For details of the suboptions, see theDiagnostics Guide. See also **-Xnojit**. By default, the JIT is enabled.

-Xlinenumbers
Displays line numbers in stack traces, for debugging. See also **-Xnolinenumbers**. By default, line numbers are on.

-Xloa Allocates a large object area (LOA). Objects will be allocated in this LOA rather than the SOA. By default, the LOA is enabled for all GC policies except for subpool, where the LOA is not available. See also **-Xnoloa**.

-Xloainitial <*number*>
<number> is between 0 and 0.95, which specifies the initial percentage of the current tenure space allocated to the large object area (LOA). The default is 0.05 or 5%.

-Xloamaximum <*number*>
<number> is between 0 and 0.95, which specifies the maximum percentage of the current tenure space allocated to the large object area(LOA). The default is 0.5 or 50%.

-Xmca<*size*>
Sets the expansion step for the memory allocated to store the RAM portion of loaded classes. Each time more memory is required to store classes in RAM, the allocated memory is increased by this amount. By default, the expansion step is 32 KB.

-Xmco<*size*>
Sets the expansion step for the memory allocated to store the ROM portion of loaded classes. Each time more memory is required to store classes in ROM, the allocated memory is increased by this amount. By default, the expansion step is 128 KB.

-Xmso<*size*>
Sets the C stack size for forked Java threads. By default, this option is set to 32 KB on 32-bit platforms and 256 KB on 64-bit platforms.

-Xmx<*size*>
Sets maximum Java heap size. By default, this option is set internally according to your system's capability.

-Xnoaot
Disables the AOT (Ahead-of-time) compiler. See also **-Xaot**. By default, the AOT compiler is enabled, but it is only active in conjunction with **-Xshareclasses**.

-Xnojit
Disables the JIT compiler. See also **-Xjit**. By default, the JIT compiler is enabled.

-Xnolinenumbers
Disables the line numbers for debugging. See also **-Xlinenumbers**. By default, line number are on.

-Xnoloa
Prevents allocation of a large object area (LOA). All objects will be allocated in the SOA. By default, the LOA is enabled for all GC policies except for subpool, where the LOA is not available. See also **-Xloa**.

-Xnosigcatch
Disables JVM signal handling code. See also **-Xsigcatch**. By default, signal handling is enabled.

-Xnosigchain
Disables signal handler chaining. See also **-Xsigchain**. By default, the signal handler chaining is enabled.

-Xoptionsfile=<*file*>
Specifies a file that contains JVM options and defines. By default, no option file is used.

-Xoss<*size*>
Sets the Java stack size and C stack size for any thread. This option is provided for compatibility and is equivalent to setting both **-Xss** and **-Xmso** to the specified value.

-Xquickstart
Improves startup time by delaying JIT compilation and optimizations. By default, quickstart is disabled and there is no delay in JIT compilation.

-Xrdbginfo:<*host*>**:**<*port*>
Loads and passes options to the remote debug information server. By default, the remote debug information server is disabled.

-Xrs Disables signal handling in the JVM. Setting **-Xrs** prevents the Java runtime from handling any internally or externally generated signals such as SIGSEGV and SIGABRT. Any signals raised are handled by the default operating system handlers. For more information on how the VM makes full use of operating system signals, see the Diagnostics Guide.

-Xrun<*library name*>**[:options]**
Loads helper libraries. To load multiple libraries, specify it more than once on the command line. Examples of these libraries are:

-Xrunhprof[:help] | [:<*option*>**=**<*value*>**, ...]**
Performs heap, CPU, or monitor profiling. For more information, see the Diagnostics Guide.

-Xrunjdwp[:help] | [:<*option*>**=**<*value*>**, ...]**
Loads debugging libraries to support the remote debugging of applications. This is the same as **-Xdbg**. For more information, see the Diagnostics Guide.

-Xrunjnichk[:help] | [:<*option*>**=**<*value*>**, ...]**
Performs additional checks for JNI functions, to trace errors in native programs that access the JVM using JNI. For more information, see the Diagnostics Guide.

-Xscmx<*size*>**[k | m | g]**
For details of **-Xscmx**, see "Class data sharing command-line options" on page 400.

-Xsigcatch
Enables VM signal handling code. See also **-Xnosigcatch**. By default, signal handling is enabled

-Xsigchain
Enables signal handler chaining. See also **-Xnosigchain**. By default, signal handler chaining is enabled.

-Xsoftrefthreshold<*number*>
Sets the number of GCs after which a soft reference will be cleared if its referent has not been marked. The default is 3, meaning that on the third GC where the referent is not marked the soft reference will be cleared.

-Xss<*size*>
Sets the maximum Java stack size for any thread. By default, this option is set to 256 KB. For more information, see the Diagnostics Guide.

-Xthr:<*options*>
Sets the threading options.

-Xverify
Enables strict class checking for every class that is loaded. By default, strict class checking is disabled.

-Xverify:none
Disables strict class checking. By default, strict class checking is disabled.

System properties

System properties are available to applications, and help provide information about the runtime environment.

com.ibm.jvm.realtime
This property enables Java applications to determine if they are running within a WebSphere Real Time for RT Linux environment.

If your application is running within the IBM WebSphere Real Time for RT Linux runtime, and was started with the **-Xrealtime** option, the **com.ibm.jvm.realtime** property has the value "hard".

If your application is running within the IBM WebSphere Real Time for RT Linux runtime, but was not started with the **-Xrealtime** option, the **com.ibm.jvm.realtime** property is not set.

If your application is running within the IBM WebSphere Real Time runtime, the **com.ibm.jvm.realtime** property has the value "soft".

Default settings for the JVM

Default settings apply to the Real Time JVM when no changes are made to the environment that the JVM runs in. Common settings are shown for reference.

Default settings can be changed using environment variables or command-line parameters at JVM startup. The table shows some of the common JVM settings. The last column indicates how you can change the behavior, where the following keys apply:

- **e** - setting controlled by environment variable only
- **c** - setting controlled by command-line parameter only
- **ec** - setting controlled by both environment variable and command-line parameter, with command-line parameter taking precedence.

The information is provided as a quick reference and is not comprehensive.

JVM setting	Default	Setting affected by
Javadumps	Enabled	ec
Javadumps on out of memory	Enabled	ec
Heapdumps	Disabled	ec
Heapdumps on out of memory	Enabled	ec
Sysdumps	Enabled	ec
Where dump files are produced	Current directory	ec
Verbose output	Disabled	c
Boot classpath search	Disabled	c
JNI checks	Disabled	c
Remote debugging	Disabled	c

JVM setting	Default	Setting affected by
Strict conformancy checks	Disabled	c
Quickstart	Disabled	c
Remote debug info server	Disabled	c
Reduced signalling	Disabled	c
Signal handler chaining	Enabled	c
Classpath	Not set	ec
Class data sharing	Disabled	c
Accessibility support	Enabled	e
JIT compiler	Enabled	ec
AOT compiler (AOT is not used by the JVM unless shared classes are also enabled)	Enabled	c
JIT debug options	Disabled	c
Java2D max size of fonts with algorithmic bold	14 point	e
Java2D use rendered bitmaps in scalable fonts	Enabled	e
Java2D freetype font rasterizing	Enabled	e
Java2D use AWT fonts	Disabled	e
Default locale	None	e
Time to wait before starting plug-in	zero	e
Temporary directory	/tmp	e
Plug-in redirection	None	e
IM switching	Disabled	e
IM modifiers	Disabled	e
Thread model	N/A	e
Initial stack size for Java Threads 32-bit. Use: **-Xiss<size>**	2 KB	c
Maximum stack size for Java Threads 32-bit. Use: **-Xss<size>**	256 KB	c
Stack size for OS Threads 32-bit. Use **-Xmso<size>**	256 KB	c
Initial heap size. Use **-Xms<size>**	64 MB	c
Maximum Java heap size. Use **-Xmx<size>**	Half the available memory with a minimum of 16 MB and a maximum of 512 MB	c
Target time interval utilization for an application. The Garbage collector attempts to use the remainder. Use **-Xgc:targetUtilization=<percentage>**	70%	c
The number of garbage collector threads to run. Use **-Xgc:threads=<value>**	1	c

Note: "available memory" is the smallest of real (physical) memory and the **RLIMIT_AS** value.

WebSphere Real Time for RT Linux class libraries

A reference to the Java class libraries that are used by WebSphere Real Time for RT Linux.

The Java class libraries that are used by WebSphere Real Time for RT Linux are described in http://www.rtsj.org/specjavadoc/book_index.html.

Related concepts

Chapter 7, “Support for RTSJ,” on page 61
WebSphere Real Time for RT Linux implements the Real-Time Specification for Java (RTSJ).

Running with TCK

If you are running the Real-Time Specification for Java (RTSJ) Technology Compatibility Kit (TCK) with WebSphere Real Time for RT Linux, you should include `demo/realtime/TCKibm.jar` in the classpath in order for tests to be completed successfully.

`TCKibm.jar` includes the class **VibmcorProcessorLock** which is IBM's extension to the TCK.ProcessorLock class. This class provides uniprocessor behavior that is required in a small set of TCK tests. For more information on the TCK.ProcessorLock class and vendor specific extensions to this class, see the readme file that is included with the TCK distribution.

Chapter 16. Developing WebSphere Real Time for RT Linux applications using Eclipse

Using Eclipse provides you with a fully-featured IDE when developing your real-time applications.

Before you begin

If this is the first time that you have used the Eclipse application development environment to develop real-time applications, use this procedure to configure your environment.

WebSphere Real Time for RT Linux supplies the standard Sun javac compiler. There are no restrictions on which compiler you use, but it must produce valid Java 5.0 class files. However, the javax.realtime.* Java classes have to be on the build path.

About this task

To develop your applications on Eclipse, follow these instructions:

Procedure

1. Download Eclipse from http://www.eclipse.org/downloads/. It is recommended that you use Eclipse 3.1.2 for correct Java 5.0 compilation.
2. Download IBM SDK and Runtime Environment for Linux platforms, Java 2 Technology Edition, Version 5.0 compliant JVM for running Eclipse.
3. Extract this file, `/opt/ibm/ibm-wrt-i386-60/jre/lib/i386/realtime/jclSC160/realtime.jar`, from the WebSphere Real Time for RT Linux package.
4. Open Eclipse and create a project. Click **File** → **New**. Select **Java project** from the **New Project** panel.

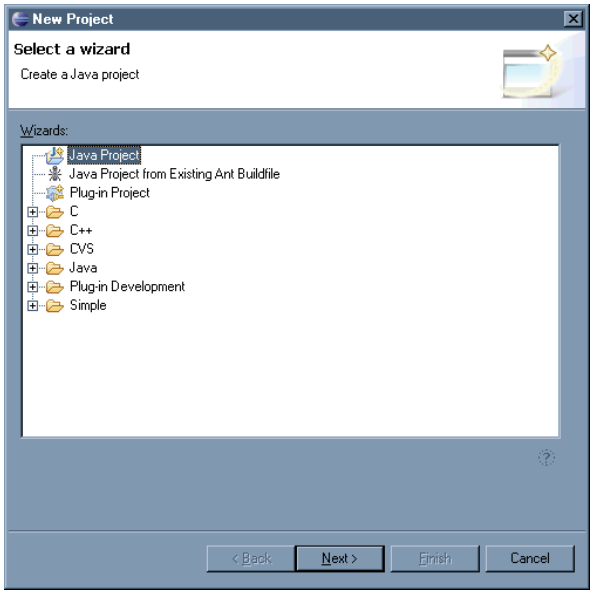


Figure 15. New Project panel

5. Click **Next** to display the **New Java Project** panel.

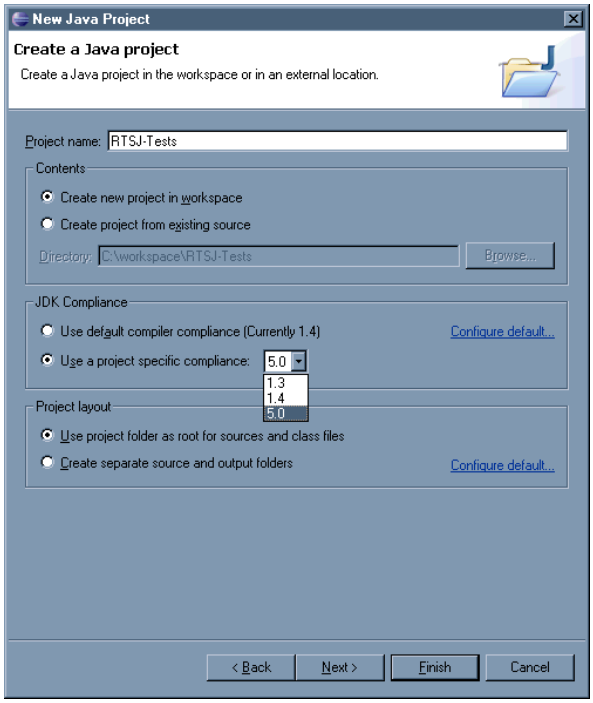


Figure 16. New Java Project panel

   a. Enter a project name, for example, RTSJ-Tests.
   b. Check that the JDK compiler is set to 5.0.
6. Click **Finish**.
7. Create a working directory and import the `/opt/ibm/ibm-wrt-i386-60/jre/lib/i386/realtime/jclSC160/realtime.jar` file.

8. Click **File** → **New** → **Folder** to open the **New Folder** panel. Enter a new folder name, for example, *deplib*.

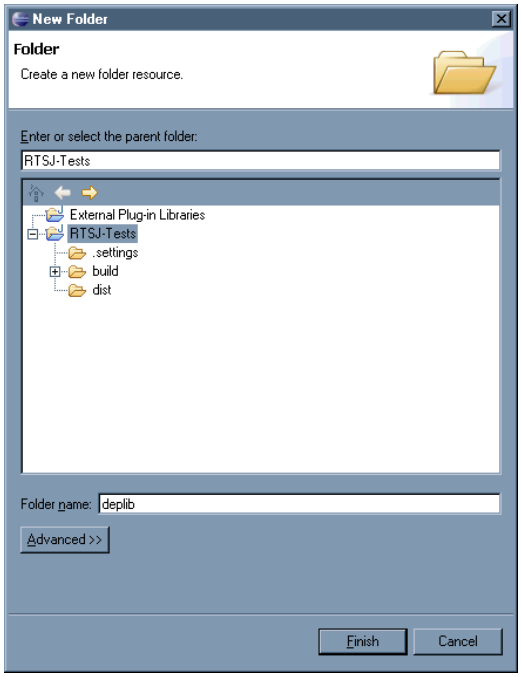


Figure 17. New Folder panel

9. Click **Finish**.
10. To import your `realtime.jar` file, click **File** → **Import** to open the **Import** panel.

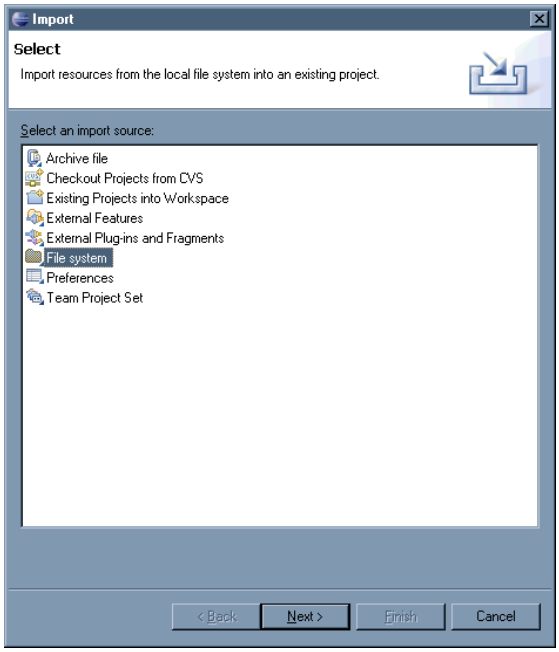


Figure 18. Import - Select panel

11. Click **File System** and click **Next**.

12. Open the `/opt/ibm/ibm-wrt-i386-60/jre/lib/i386/realtime/jclSC160/` directory on the filesystem where the JVM was unpacked.

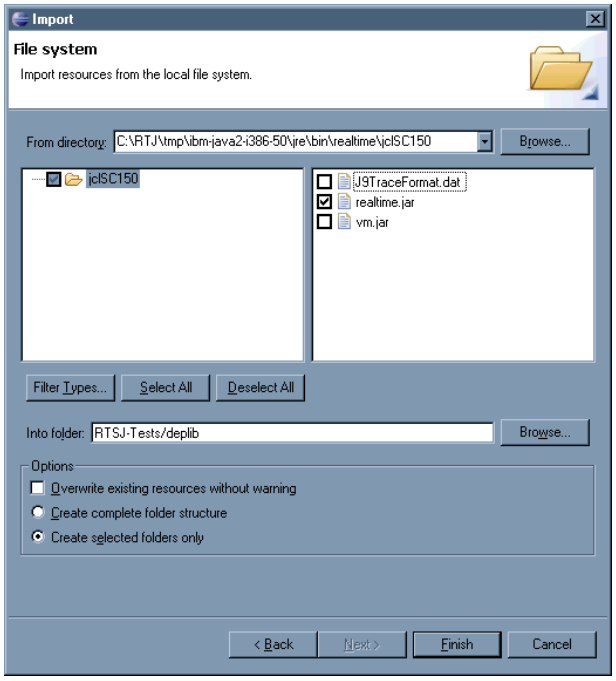


Figure 19. Import - file system panel

13. Click **Finish**.
14. Add the jar file to the library path. Right-click your project and click **Properties** to open the **Properties** panel.

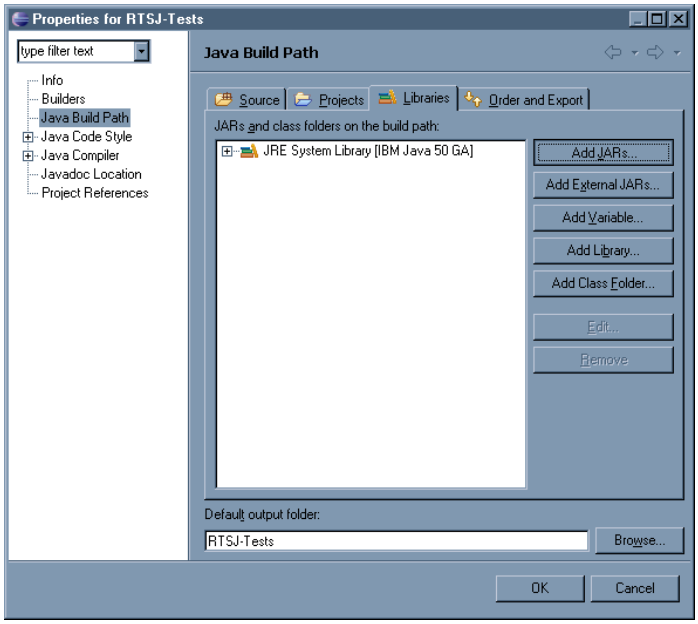


Figure 20. Properties panel

15. Click **Java Build Path** and the **Libraries** tab, as shown in Figure 20. Click **Add Jars**.

16. Click **realtime.jar** under your project directory. Click **OK**.

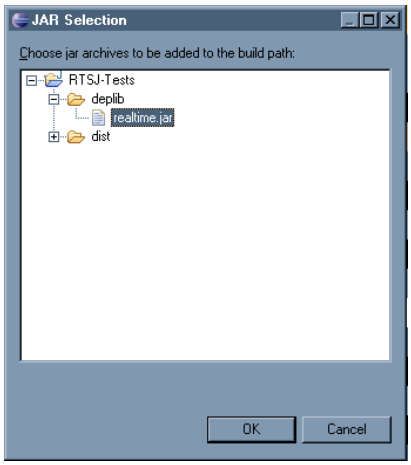


Figure 21. JAR Selection panel

Results

If this procedure was successful, your properties panel will look like Figure 22.

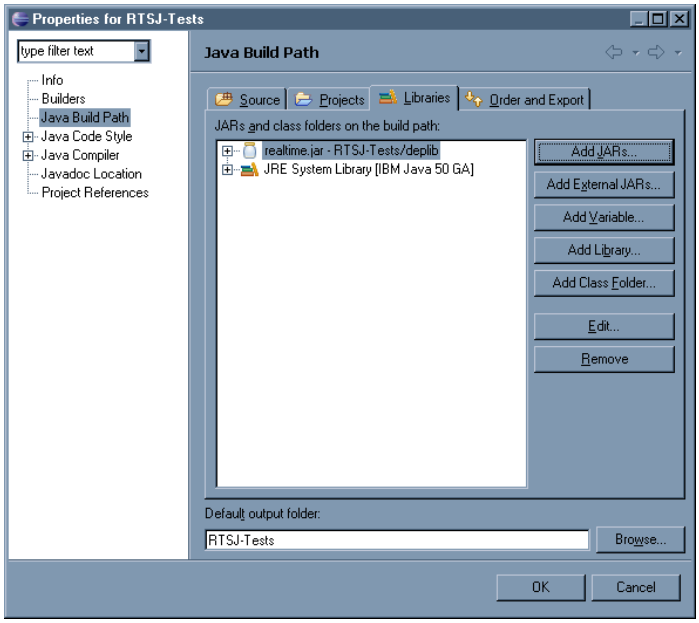


Figure 22. Properties panel

Example

Eclipse can use the `realtime.src.jar` to present additional information on the RTSJ classes. To do this, open the properties window (see Figure 23 on page 352) for the imported `realtime.jar` file, click **Java Source Attachment** and enter in **Location path:** the location of the `realtime.src.jar` file.

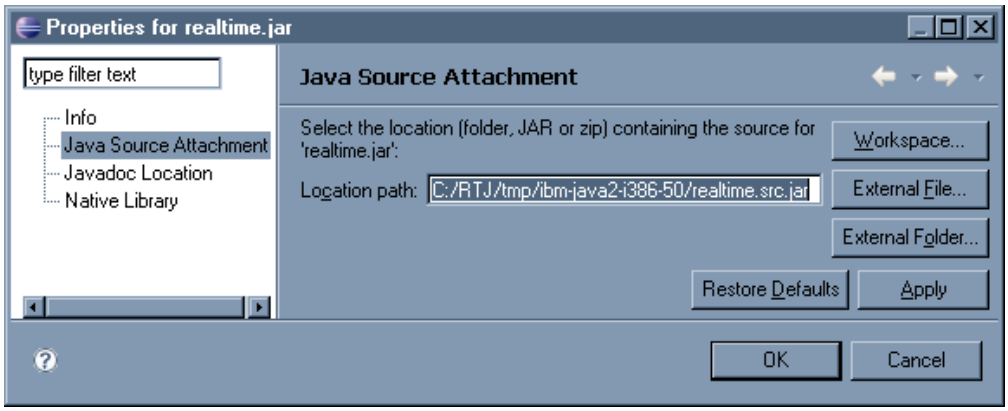


Figure 23. Properties for realtime.src.jar

What to do next

In addition to build with ant and Eclipse you have to have `realtime.jar` in your classpath in your ant build script, for example,

```
<property name="rtsj.src" location="." />
<property name="rtsj.deplib" location="deplib" />
<property name="rtsj.jar.dir" location="build/rtsj-jar.dir" />

<!-- Generate .class files for this package -->
<target name="compile" depends="init">
<javac destdir="${rtsj.jar.dir}"
srcdir="${rtsj.src}"
target="1.5"
classpath="${rtsj.deplib}/realtime.jar:${rtsj.src}"
debug="true"/>
</target>
```

This is only a part of an ant build script.

Related concepts

"Useful tools" on page 10
You can use these tools with WebSphere Real Time for RT Linux. Some of these tools are in the early stages of development and might not be fully supported.

Debugging your applications

Using the Eclipse Application developer you can debug your applications either locally or remotely.

About this task

To debug your real-time application remotely, the JVM being debugged requires the following option.

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=10100
```

Procedure

1. In the Linux environment where your application is running, enter:

   ```
   java -Xrealtime -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=10100
   ```

   where:
   - `server=y` indicates that the JVM is accepting connections from debuggers.
   - `suspend=y` makes the JVM wait for a debugger to attach before running.

- `address=10100` is the port number to which the debugger should attach to the JVM. This number should normally be above 1024.

The JVM displays the following message:

```
Listening for transport dt_socket at address: 10100
```

2. Open your application in Eclipse and select **Debug**.

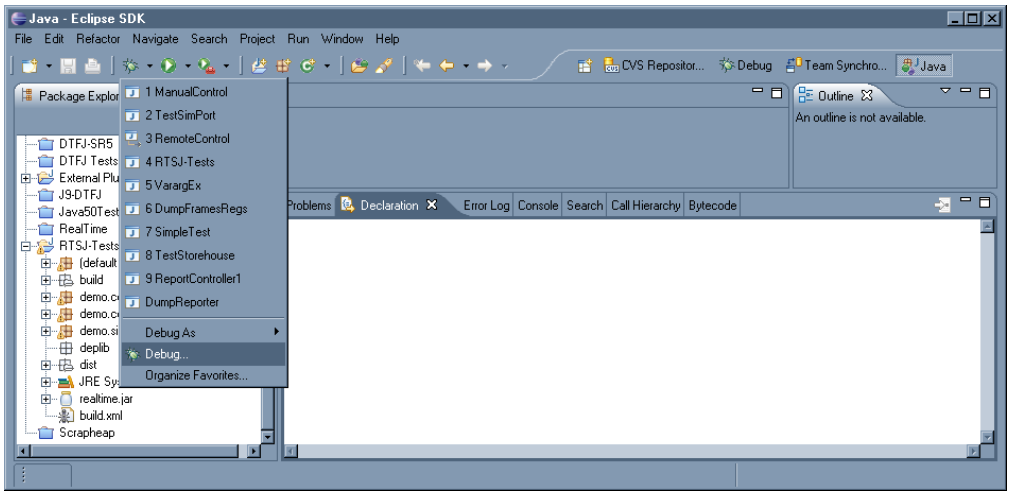


Figure 24. Java - Eclipse panel

3. A new configuration for debugging remote applications should be created. You need only to create one if an application in the same project is run, and is listening on the same port for each run.

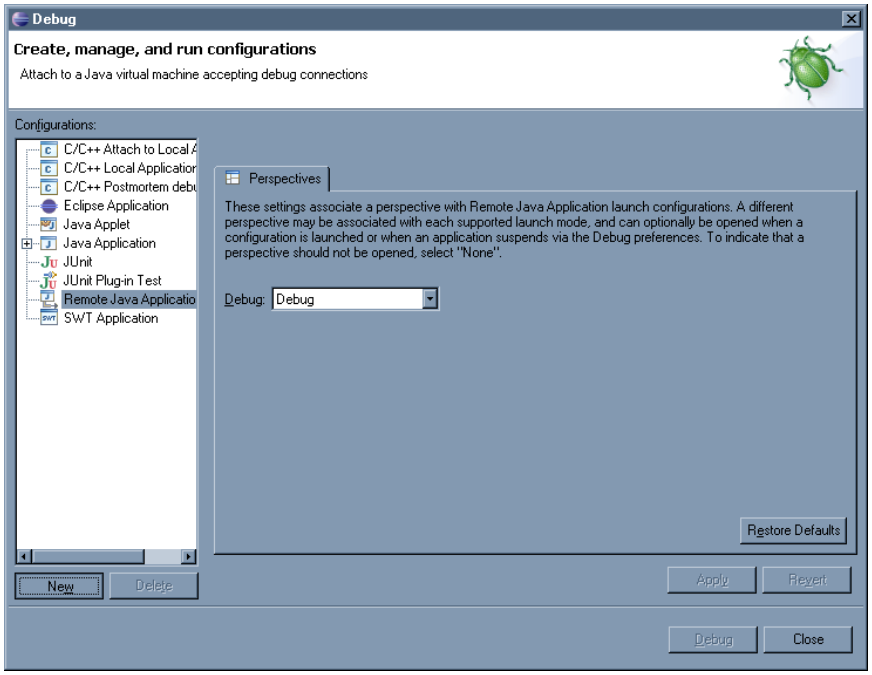


Figure 25. Debug panel

4. When you have created the configuration, fill in the name of the Configurations (RemoteApp in this case), the name of the project that contains the application you are debugging, the *hostname* of the workstation where the application is

running, and the port number you passed in the **-agentlib** options.

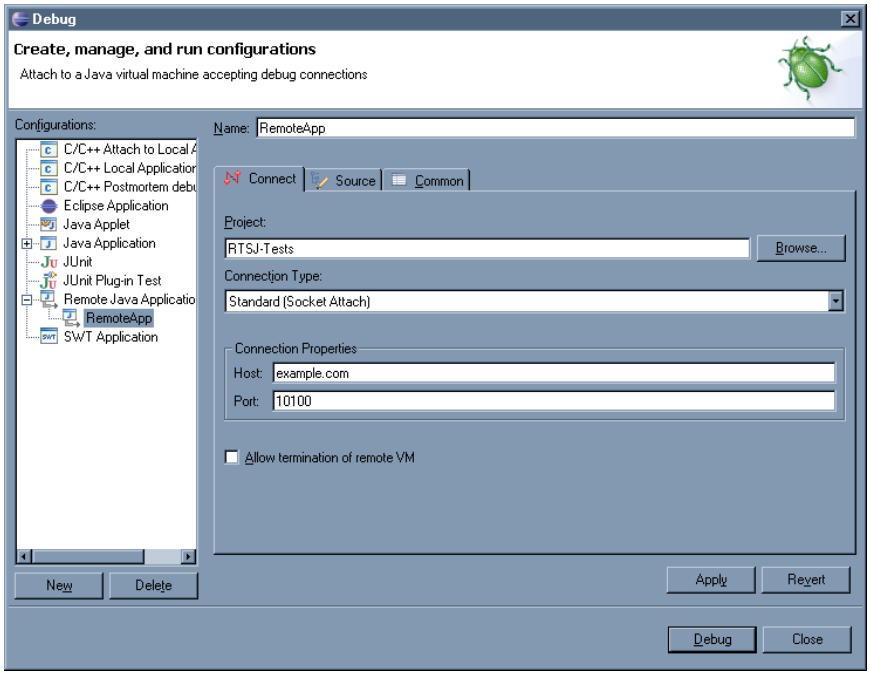


Figure 26. Debug panel

5. Click **Debug** to start the debugging session. The **Debug** perspective should be open for you to view the state of the remotely debugged JVM.

Running Eclipse with the Realtime JVM

This section explains how to run Eclipse with the Realtime JVM.

To run Eclipse with the Realtime JVM you need to specify on the "eclipse" command:

- the fully qualified directory to the java executable of the Realtime JVM you intend to use
- the **-Xrealtime** JVM option
- the size of the Immortal Memory you want Eclipse to use. This should be at least 128M.

Example of running eclipse with the Realtime JVM:

```
eclipse -vm $JAVA_HOME/jre/bin/java -vmargs -Xrealtime -Xgc:immortalMemorySize=128M
```

Note: The Eclipse SDK does not take advantage of the various Realtime memory options which are available to Realtime applications, a consequence of this, is for Immortal Memory to become exhausted, especially in cases where Eclipse is used for many hours or days without it being restarted. If an **OutOfMemory** error does occur, you can increase the value on the **-Xgc:immortalMemorySize** option to increase the amount of Immortal Memory you want Eclipse to use.

Appendix A. User Guide

IBM SDK and Runtime Environment for Linux platforms, Java Technology Edition, Version 6, User Guide

Copyright information

This edition of the user guide applies to the IBM SDK and Runtime Environment for Linux on multiple platforms.

The platforms this guide applies to are:

- IBM SDK and Runtime Environment for Linux platforms, Java Technology Edition, Version 6

and all subsequent releases and modifications until otherwise indicated in new editions.

Preface

This user guide provides general information about the IBM SDK and Runtime Environment for Linux platforms, Java Technology Edition, Version 6. The user guide gives specific information about any differences in the IBM implementation compared with the Sun implementation.

Read this user guide with the more extensive documentation on the Sun Web site: http://java.sun.com.

For the list of distributions against which the SDK and Runtime Environment for Linux have been tested, see: http://www.ibm.com/developerworks/java/jdk/linux/tested.html.

(Intel 32-bit platforms only) These Virtualized Environments are supported:

- VMWare
- Xen
- Microsoft Virtual Server

The Diagnostics Guide provides more detailed information about the IBM Virtual Machine for Java.

This user guide is part of a release and is applicable only to that particular release. Make sure that you have the user guide appropriate to the release you are using.

The terms "Runtime Environment" and "Java Virtual Machine" are used interchangeably throughout this user guide.

Overview

The IBM SDK is a development environment for writing and running applets and applications that conform to the Java 6 Core Application Program Interface (API).

The SDK includes the Runtime Environment for Linux, which enables you only to run Java applications. If you have installed the SDK, the Runtime Environment is included.

The Runtime Environment contains the Java Virtual Machine and supporting files including non-debuggable .so files and class files. The Runtime Environment contains only a subset of the classes that are found in the SDK and allows you to support a Java program at runtime but does not provide compilation of Java programs. The Runtime Environment for Linux does not include any of the development tools, for example **appletviewer** or the Java compiler (**javac**), or classes that are only for development systems.

In addition, for IA32, PPC32/PPC64, and AMD64/EM64T platforms, the Java Communications application programming interface (API) package is provided for use with the Runtime Environment for Linux. You can find information about it in "Java Communications API (JavaComm)" on page 407.

The `license_xx.html` file contains the license agreement for the Runtime Environment for Linux software, where *xx* is an abbreviation for the language. To view or print the license agreement, open the file in a Web browser.

Version compatibility

In general, any applet or application that ran with a previous version of the SDK should run correctly with the IBM SDK for Linux, v6. Classes compiled with this release are not guaranteed to work on previous releases.

For information about compatibility issues between releases, see the Sun Web site at:

http://java.sun.com/javase/6/webnotes/compatibility.html

http://java.sun.com/j2se/5.0/compatibility.html

http://java.sun.com/j2se/1.4/compatibility.html

http://java.sun.com/j2se/1.3/compatibility.html

If you are using the SDK as part of another product (for example, IBM WebSphere Application Server), and you upgrade from a previous level of the SDK, perhaps v5.0, serialized classes might not be compatible. However, classes are compatible between service refreshes.

Migrating from other IBM JVMs

From Version 5.0, the IBM Runtime Environment for Linux contains new versions of the IBM Virtual Machine for Java and the Just-In-Time (JIT) compiler.

If you are migrating from an older IBM Runtime Environment, note that:

- The JVM shared library `libjvm.so` is now stored in jre/lib/<arch>/j9vm and jre/lib/<arch>/classic.
- From Version 5.0 onwards, the JVM Monitoring Interface (JVMMI) is no longer available. You must rewrite JVMMI applications to use the JVM Tool Interface (JVMTI) instead. The JVMTI is not functionally the equivalent of JVMMI. For information about JVMTI, see http://java.sun.com/javase/6/docs/technotes/guides/jvmti/ and the Diagnostics Guide.

- From Version 5.0 onwards, the implementation of JNI conforms to the JNI specification, but differs from the Version 1.4.2 implementation. It returns copies of objects rather than pinning the objects. This difference can expose errors in JNI application code. For information about debugging JNI code, see **-Xcheck:jni** in "JVM command-line options" on page 415.
- From Version 5.0 onwards, the format and content of garbage collector verbose logs obtained using **-verbose:gc** have changed. The data is now formatted as XML. The data content reflects the changes to the implementation of garbage collection in the JVM, and most of the statistics that are output have changed. You must change any programs that process the verbose GC output so that they will work with the new format and data. See the Diagnostics Guide for an example of the new verbose GC data.
- SDK 1.4 versions of the IBM JRE included JVM specific classes in a file called core.jar. From Version 5.0 onwards, these are included in a file called `vm.jar`.
- From Version 6, JVM classes are held in multiple JAR files in the `jre/lib` directory. This replaces the single `rt.jar` and `core.jar` from earlier releases.
- For additional industry compatibility information, see Sun's Java 6 Compatibility Documentation: http://java.sun.com/javase/6/webnotes/compatibility.html
- For additional deprecated API information, see Sun's Java 6 Deprecated API List: http://java.sun.com/javase/6/docs/api/deprecated-list.html
- Tracing class dependencies, started using **-verbose:Xclassdep**, is not supported. If you specify **-verbose:Xclassdep**, the JVM will issue an error message and will not start.
- The JVM detects the operating system locale and sets the language preferences accordingly. For example, if the locale is set to **fr_FR**, JVM messages will be printed in French. To avoid seeing JVM messages in the language of the detected locale, remove the file `$SDK/jre/bin/java_xx.properties` where *xx* is the locale, such as **fr**, and the JVM will print messages in English.

Contents of the SDK and Runtime Environment

The SDK contains several development tools and a Java Runtime Environment (JRE). This section describes the contents of the SDK tools and the Runtime Environment.

Applications written entirely in Java must have **no** dependencies on the IBM SDK's directory structure (or files in those directories). Any dependency on the SDK's directory structure (or the files in those directories) might result in application portability problems.

The user guides, Javadoc files, demo files, and the accompanying license and copyright files are the only documentation included in this SDK for Linux. You can view Sun's software documentation by visiting the Sun Web site, or you can download Sun's software documentation package from the Sun Web site: http://java.sun.com.

Contents of the Runtime Environment

A list of classes, tools, and other files that you can use with the standard Runtime Environment.

- Core Classes - These classes are the compiled class files for the platform and must remain compressed for the compiler and interpreter to access them. Do not modify these classes; instead, create subclasses and override where you need to.

- Trusted root certificates from certificate signing authorities - These certificates are used to validate the identity of signed material. The IBM Runtime Environment for Java contains an expired GTE CyberTrust Certificate for compatibility reasons. This certificate might be removed for later versions of the SDK. See "Expired GTE Cybertrust Certificate" on page 441 for more information.
- JRE tools - The following tools are part of the Runtime Environment and are in the /opt/ibm/ibm-wrt-i386-60/jre/bin directory unless otherwise specified.

  ikeyman (iKeyman GUI utility)
  Allows you to manage keys, certificates, and certificate requests. For more information see the accompanying Security Guide and http://public.dhe.ibm.com/software/dw/jdk/security/50/GSK7c_SSL_IKM_Guide.pdf. The SDK also provides a command-line version of this utility.

  java (Java Interpreter)
  Runs Java classes. The Java Interpreter runs programs that are written in the Java programming language.

  javaw (Java Interpreter)
  Runs Java classes in the same way as the **java** command does, but does not use a console window.

  ***(Linux IA 32-bit, PPC32, and PPC64 only)* javaws (Java Web Start)**
  Enables the deployment and automatic maintenance of Java applications. For more information, see "Running Web Start" on page 398.

  jcontrol (Java Control Panel)
  (Except System z® platforms) Configures your Runtime Environment.

  jextract (Dump extractor)
  Converts a system-produced dump into a common format that can be used by jdmpview. For more information, see jdmpview.

  keytool (Key and Certificate Management Tool)
  Manages a keystore (database) of private keys and their associated X.509 certificate chains that authenticate the corresponding public keys.

  kinit
  Obtains and caches Kerberos ticket-granting tickets.

  klist
  Displays entries in the local credentials cache and key table.

  ktab
  Manages the principal names and service keys stored in a local key table.

  pack200
  Transforms a JAR file into a compressed pack200 file using the Java gzip compressor.

  policytool (Policy File Creation and Management Tool)
  Creates and modifies the external policy configuration files that define your installation's Java security policy.

  rmid (RMI activation system daemon)
  Starts the activation system daemon so that objects can be registered and activated in a Java virtual machine (JVM).

  rmiregistry (Java remote object registry)
  Creates and starts a remote object registry on the specified port of the current host.

tnameserv (Common Object Request Broker Architecture (CORBA) transient naming service)
Starts the CORBA transient naming service.

unpack200
Transforms a packed file produced by pack200 into a JAR file.

Contents of the SDK

A list of tools and reference information that is included with the standard SDK.

The following tools are part of the SDK and are located in the /opt/ibm/ibm-wrt-i386-60/bin directory:

appletviewer (Java Applet Viewer)
Tests and runs applets outside a Web browser.

apt (Annotation Processing Tool)
Finds and executes annotation processors based on the annotations present in the set of specified source files being examined.

ControlPanel (Java Control Panel)
(Except System z platforms) Configures your Runtime Environment.

extcheck (Extcheck utility)
Detects version conflicts between a target jar file and currently-installed extension jar files.

***(Linux IA 32-bit and PPC32)* HtmlConverter (Java Plug-in HTML Converter)**
Converts an HTML page that contains applets to a format that can use the Java Plug-in.

idlj (IDL to Java Compiler)
Generates Java bindings from a given IDL file.

ikeycmd (iKeyman command-line utility)
Allows you to manage keys, certificates, and certificate requests from the command line. For more information see the accompanying *Security Guide* and http://www.ibm.com/developerworks/java/jdk/security/.

jar (Java Archive Tool)
Combines multiple files into a single Java Archive (JAR) file.

jarsigner (JAR Signing and Verification Tool)
Generates signatures for JAR files and verifies the signatures of signed JAR files.

java (Java Interpreter)
Runs Java classes. The Java Interpreter runs programs that are written in the Java programming language.

java-rmi.cgi (HTTP-to-CGI request forward tool)
Accepts RMI-over-HTTP requests and forwards them to an RMI server listening on any port.

javac (Java Compiler)
Compiles programs that are written in the Java programming language into bytecodes (compiled Java code).

javadoc (Java Documentation Generator)
Generates HTML pages of API documentation from Java source files.

javah (C Header and Stub File Generator)
Enables you to associate native methods with code written in the Java programming language.

javap (Class File Disassembler)
Disassembles compiled files and can print a representation of the bytecodes.

javaw (Java Interpreter)
Runs Java classes in the same way as the **java** command does, but does not use a console window.

***(Linux IA 32-bit, PPC32, and PPC64 only)* javaws (Java Web Start)**
Enables the deployment and automatic maintenance of Java applications. For more information, see "Running Web Start" on page 398.

jconsole (JConsole Monitoring and Management Tool)
Monitors local and remote JVMs using a GUI. JMX-compliant.

jdb (Java Debugger)
Helps debug your Java programs.

jdmpview (Cross-platform dump formatter)
Analyzes dumps. For more information, see the Diagnostics Guide.

keytool (Key and Certificate Management Tool)
Manages a keystore (database) of private keys and their associated X.509 certificate chains that authenticate the corresponding public keys.

native2ascii (Native-To-ASCII Converter)
Converts a native encoding file to an ASCII file that contains characters encoded in either Latin-1 or Unicode, or both.

policytool (Policy File Creation and Management Tool)
Creates and modifies the external policy configuration files that define your installation's Java security policy.

rmic (Java Remote Method Invocation (RMI) Stub Converter)
Generates stubs, skeletons, and ties for remote objects. Includes RMI over Internet Inter-ORB Protocol (RMI-IIOP) support.

rmid (RMI activation system daemon)
Starts the activation system daemon so that objects can be registered and activated in a Java virtual machine (JVM).

rmiregistry (Java remote object registry)
Creates and starts a remote object registry on the specified port of the current host.

schemagen
Creates a schema file for each namespace referenced in your Java classes.

serialver (Serial Version Command)
Returns the serialVersionUID for one or more classes in a format that is suitable for copying into an evolving class.

tnameserv (Common Object Request Broker Architecture (CORBA) transient naming service)
Starts the CORBA transient naming service.

wsgen
Generates JAX-WS portable artifacts used in JAX-WS Web services.

wsimport
Generates JAX-WS portable artifacts from a Web Services Description Language (WSDL) file.

xjc
Compiles XML Schema files.

Include Files
C headers for JNI programs.

Demos
The `demo` directory contains a number of subdirectories containing sample source code, demos, applications, and applets that you can use. From Version 6, the RMI-IIOP demonstration is not included with the SDK.

copyright
The copyright notice for the SDK for Linux software.

License
The License file, /opt/ibm/ibm-wrt-i386-60/docs/<*locale*>/license_<*locale*>.html, contains the license agreement for the SDK for Linux software (where <*locale*> is the name of your locale, for example en). To view or print the license agreement, open the file in a Web browser.

Installing and configuring the SDK and Runtime Environment

You can install the IBM Java SDK and Runtime Environment from either an RPM or a .tgz file. Unless you want to allow all the users on the workstation to access this Java installation, use the .tgz installation method. If you do not have root access, use the .tgz file.

If you install using an RPM file, the Java files are installed in /opt/ibm/ibm-wrt-i386-60/. The examples in this guide assume that you have installed Java in this directory.

Upgrading the SDK

If you are upgrading the SDK from a previous release, back up all the configuration files and security policy files before you start the upgrade.

What to do next

After the upgrade, you might have to restore or reconfigure these files because they might have been overwritten during the upgrade process. Check the syntax of the new files before restoring the original files because the format or options for the files might have changed.

Installing on Red Hat Enterprise Linux (RHEL) 4

The SDK depends on shared libraries that are not installed by default for Red Hat Enterprise Linux (RHEL).

About this task

In RHEL 4, the RPMs that contain these libraries are:

- `compat-libstdc++-33-3.2.3` and `xorg-x11-deprecated-libs-6.8.1` (Platforms other than zSeries)
- `compat-libstdc++-295-2.95.3` and `xorg-x11-deprecated-libs-6.8.1` (zSeries)

To include these libraries during RHEL 4 installation:

Procedure

1. When you reach the **Package Defaults** screen, select **Customize the set of packages to be installed**.
2. At the **Package Group Selection** screen, under **X Windows System**, choose **Details** and make sure that you have selected `xorg-x11-deprecated-libs`
3. Under the **Development** options, select **Legacy Software Development**.

Installing on Red Hat Enterprise Linux (RHEL) 5

The SDK depends on shared libraries that are not installed by default for Red Hat Enterprise Linux (RHEL).

About this task

In RHEL 5, these RPMs contain the shared libraries:

- `compat-libstdc++-33-3.2.3` (Platforms other than zSeries)
- `compat-libstdc++-295-2.95.3` (zSeries)

To include these libraries during RHEL 5 installation:

Procedure

1. At the software selection screen, select **Customize now**.
2. On the next screen, in the left panel, select **Base System**; in the right panel, select **Legacy Software Support**. These selections install the compat-libstdc++ packages.

Running Java with SELinux on RHEL 5

You can run the IBM SDK for Java on Red Hat Enterprise Linux Version 5 with SELinux enabled on all platforms except PPC without any restrictions. For PPC platforms, Java must be installed in the default directory or you must enable it manually.

About this task

To enable Java manually on PPC platforms, enter this command:

```
chcon -R -t texrel_shlib_t <path_of_sdk>
```

Where *<path_of_sdk>* is the path where Java is installed.

For more information about SELinux, see Introduction to SELinux in the Red Hat documentation.

Installing a 32-bit SDK on 64-bit architecture

To run the SDK, you must install the correct versions of all libraries required by the SDK, either 32- or 64-bit.

About this task

In RHEL4, the 64-bit versions of the packages are available in the **Compatibility Arch Support** package group.

You can use the RPM tool to check which versions of the packages you have installed by adding the option `--queryformat "%{NAME}.%{ARCH}\n"` to your RPM command. For example:

```
/home/username : rpm --queryformat "%{NAME}.%{ARCH}\n" -q libstdc++
libstdc++.x86_64
libstdc++.i386
```

Installing from a `.rpm` file

A procedure for installing from a `.rpm` file.

About this task

To upgrade your JVM using the rpm tool, you must uninstall any previous version. To install two versions of the JVM in different locations, use the rpm **--force** option to ignore the version conflict. Alternatively, install the JVM from the `.tgz` file.

Procedure

1. Open a shell prompt, making sure that you are root.
2. At a shell prompt, type `rpm -ivh <RPM file>`. For example:

   ```
   rpm -ivh ibm-ws-rt-i386-sdk-1.0-1.0.i386.rpm
   ```

   or

   ```
   rpm -ivh ibm-ws-rt-i386-jre-1.0-1.0.i386.rpm
   ```

Installing from a `.tgz` file

A procedure for installing from a `.tgz` file.

Procedure

1. Create a directory to store the Java package files. The examples in this guide assume that you have installed in /opt/ibm/ibm-wrt-i386-60/. In the rest of the guide, replace /opt/ibm/ibm-wrt-i386-60/ with the directory in which you installed Java.
2. At a shell prompt, type `tar -zxvf <.tgz file>`.

   ```
   tar -zxvf ibm-ws-rt-sdk-1.0-1.0-linux-i386.tgz
   ```

   or

   ```
   tar -zxvf ibm-ws-rt-jre-1.0-1.0-linux-i386.tgz
   ```

3. If you are running Security-Enhanced Linux (SELinux), you must identify the Java shared libraries to the system. Type:

   ```
   chcon -R -t texrel_shlib_t /opt/ibm/ibm-wrt-i386-60/jre
   chcon -R -t texrel_shlib_t /opt/ibm/ibm-wrt-i386-60/bin
   chcon -R -t texrel_shlib_t /opt/ibm/ibm-wrt-i386-60/lib
   ```

Configuring the SDK and Runtime Environment for Linux

Inconsistencies in the font encodings on Red Hat Advanced Server

Note: ***(For Linux IA 32-bit Chinese users only)*** Because of inconsistencies in the font encodings on Red Hat Advanced Server, when you install for an environment in which you want Chinese to be the default language, it is better to install with a default language of English and then change to Chinese after the installation is complete. Otherwise, you might find that the Chinese fonts do not display.

Setting the path

If you alter the **PATH** environment variable, you will override any existing Java launchers in your path.

About this task

The **PATH** environment variable enables Linux to find programs and utilities, such as javac, java, and javadoc tool, from any current directory. To display the current value of your **PATH**, type the following at a command prompt:

```
echo $PATH
```

To add the Java launchers to your path:

1. Edit the shell startup file in your home directory (typically .bashrc, depending on your shell) and add the absolute paths to the **PATH** environment variable; for example:

   ```
   export PATH=/opt/ibm/ibm-wrt-i386-60/bin:/opt/ibm/ibm-wrt-i386-60/jre/bin:$PATH
   ```

2. Log on again or run the updated shell script to activate the new **PATH** environment variable.

Results

After setting the path, you can run a tool by typing its name at a command prompt from any directory. For example, to compile the file `Myfile.Java`, at a command prompt, type:

```
javac Myfile.Java
```

Setting the class path

The class path tells the SDK tools, such as java, javac, and the javadoc tool, where to find the Java class libraries.

About this task

You should set the class path explicitly only if:

- You require a different library or class file, such as one that you develop, and it is not in the current directory.
- You change the location of the `bin` and `lib` directories and they no longer have the same parent directory.
- You plan to develop or run applications using different runtime environments on the same system.

To display the current value of your **CLASSPATH** environment variable, type the following command at a shell prompt:

```
echo $CLASSPATH
```

If you develop and run applications that use different runtime environments, including other versions that you have installed separately, you must set the **CLASSPATH** and **PATH** explicitly for each application. If you run multiple applications simultaneously and use different runtime environments, each application must run in its own shell prompt.

Updating your SDK or JRE for daylight saving time changes

You can apply recent changes to daylight saving time using the IBM Time Zone Update Utility for Java (JTZU).

About this task

Many countries around the world use a daylight saving time (DST) convention. Typically, clocks move forward by one hour during the summer months to create more daylight hours during the afternoon and less during the morning. This practice has many implications, including the need to adjust system clocks in computer systems. Occasionally, countries change their DST start and end dates. These changes can affect the date and time functions in applications, because the original start and end dates are programmed into the operating system and in Java software. To avoid this problem you must update operating systems and Java installations with the new DST information.

The Olson time zone database is an external resource that compiles information about the time zones around the world. This database establishes standard names for time zones, such as "America/New_York", and provides regular updates to time zone information that can be used as reference data. To ensure that Java JREs and SDKs contain up to date DST information, IBM incorporates the latest Olson update into each Java service refresh. To find out which Olson time zone update is included for a particular service refresh, see https://www.ibm.com/developerworks/java/jdk/dst/olson_table.html.

If a DST change has been introduced since the last service refresh, you can use JTZU to directly update your Java installation. You can also use this tool to update your installation if you are unable to move straight to the latest service refresh. JTZU is available from IBM developerWorks using the following link: https://www.ibm.com/developerworks/java/jdk/dst/jtzu.html.

Results

After updating your Java installation with any recent DST changes, your application can handle time and date calculations correctly.

Uninstalling the SDK and Runtime Environment for Linux

The process that you use to remove the SDK and Runtime Environment for Linux depends on what type of installation you used.

See Uninstalling the Red Hat Package Manager (RPM) package or Uninstalling the compressed Tape Archive (TAR) package for instructions.

Uninstalling the Red Hat Package Manager (RPM) package

A procedure for uninstalling the Red Hat Package Manager (RPM) package.

About this task

To uninstall the SDK or Runtime Environment for Linux if you installed the installable RPM package:

Procedure

1. To check which RPM packages you have installed, enter: `rpm -qa | grep -i ws-rt`

   You will see a list of any IBM Java packages that you have installed; for example:

   ```
   ibm-ws-rt-i386-jre-1.0-0.0
   ibm-ws-rt-i386-sdk-1.0-0.0
   ```

This output tells you which packages you can uninstall, using the `rpm -e` command; for example:

```
rpm -e ibm-ws-rt-i386-jre-1.0-0.0
rpm -e ibm-ws-rt-i386-sdk-1.0-0.0
```

Alternatively, you can use a graphical tool such as kpackage or yast2

2. Remove from your **PATH** statement the directory in which you installed the SDK and Runtime Environment.
3. If you installed the Java Plug-in, remove the Java Plug-in files from the Web browser directory.

Uninstalling the IBM WebSphere Real Time for RT Linux

A list of the steps to remove the WebSphere Real Time for RT Linux that was extracted from the compressed package.

Procedure

1. Remove the WebSphere Real Time for RT Linux files from the directory in which you installed the WebSphere Real Time for RT Linux.
2. Remove from your **PATH** statement the directory in which you installed the WebSphere Real Time for RT Linux.
3. Log on again or run the updated shell script to activate the new **PATH** setting.

Running Java applications

Java applications can be started using the java launcher or through JNI. Settings are passed to a Java application using command-line arguments, environment variables, and properties files.

The java and javaw commands

An overview of the java and javaw commands.

Purpose

The java and javaw tools start a Java application by starting a Java Runtime Environment and loading a specified class.

The javaw command is identical to java, except that javaw has no associated console window. Use javaw when you do not want a command prompt window to be displayed. The javaw launcher displays a window with error information if it fails.

Usage

The JVM searches for the initial class (and other classes that are used) in three sets of locations: the bootstrap class path, the installed extensions, and the user class path. The arguments that you specify after the class name or jar file name are passed to the main function.

The java and javaw commands have the following syntax:

```
java [ options ] <class> [ arguments ... ]
java [ options ] -jar <file.jar> [ arguments ... ]
javaw [ options ] <class> [ arguments ... ]
javaw [ options ] -jar <file.jar> [ arguments ... ]
```

Parameters

[*options*]
Command-line options to be passed to the runtime environment.

<*class*>
Startup class. The class must contain a main() method.

<*file.jar*>
Name of the jar file to start. It is used only with the **-jar** option. The named jar file must contain class and resource files for the application, with the startup class indicated by the Main-Class manifest header.

[*arguments ...*]
Command-line arguments to be passed to the main() function of the startup class.

Obtaining version information

You obtain The IBM build and version number for your Java installation using the **-version** option. You can also obtain version information for all `jar` files on the class path by using the **-Xjarversion** option.

Procedure

1. Open a shell prompt.
2. Type the following command:

   ```
   java -version
   ```

   You will see information similar to:

```
java version "1.6.0-internal"
Java(TM) SE Runtime Environment (build 20070329_01)
IBM J9 VM (build 2.4, J2RE 1.6.0 IBM J9 2.4 Linux x86-32 jvmxi3260-20070326_12091 (JIT enabled)
J9VM - 20070326_12091_lHdSMR
JIT  - dev_20070326_1800
GC   - 20070319_AA)
```

   Exact build dates and versions will change.

What to do next

You can also list the version information for all available `jar` files on the class path, the boot class path, and in the extensions directory. Type the following command:

```
java -Xjarversion
```

You will see information similar to:

```
...
/opt/ibm/ibm-wrt-i386-60/jre/lib/ext/ibmpkcs11impl.jar  VERSION: 1.0 build_20070125
/opt/ibm/ibm-wrt-i386-60/jre/lib/ext/dtfjview.jar
/opt/ibm/ibm-wrt-i386-60/jre/lib/ext/xmlencfw.jar  VERSION: 1.00, 20061011  LEVEL: -20061011

...
```

The information available varies for each `jar` file and is taken from the **Implementation-Version** and **Build-Level** properties in the manifest of the `jar` file.

Specifying Java options and system properties

You can specify Java options and system properties on the command line, by using an options file, or by using an environment variable.

About this task

These methods of specifying Java options are listed in order of precedence. Rightmost options on the command line have precedence over leftmost options; for example, if you specify:

```
java -Xint -Xjit myClass
```

The **-Xjit** option takes precedence.

Procedure

1. By specifying the option or property on the command line. For example:
   ```
   java -Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump MyJavaClass
   ```
2. By creating a file that contains the options, and specifying it on the command line using **-Xoptionsfile=**<*file*>.
3. By creating an environment variable called **IBM_JAVA_OPTIONS** containing the options. For example:
   ```
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump"
   ```

Standard options

The definitions for the standard options.

See "JVM command-line options" on page 415 for information about nonstandard (-X) options.

-agentlib:<*libname*>**[=**<*options*>**]**
Loads a native agent library <*libname*>; for example **-agentlib:hprof**. For more information, specify **-agentlib:jdwp=help** and **-agentlib:hprof=help** on the command line.

-agentpath:*libname***[=**<*options*>**]**
Loads a native agent library by full path name.

-cp <*directories and .zip or .jar files separated by* **:**>
Sets the search path for application classes and resources. If **-classpath** and **-cp** are not used and the **CLASSPATH** environment variable is not set, the user class path is, by default, the current directory (.).

-classpath <*directories and .zip or .jar files separated by* **:**>
Sets the search path for application classes and resources. If **-classpath** and **-cp** are not used and the **CLASSPATH** environment variable is not set, the user class path is, by default, the current directory (.).

-D<*property name*>**=**<*value*>
Sets a system property.

-help or -?
Prints a usage message.

-javaagent:<*jarpath*>**[=**<*options*>**]**
Load a Java programming language agent. For more information, see the java.lang.instrument API documentation.

-jre-restrict-search
Include user private JREs in the version search.

-no-jre-restrict-search
Exclude user private JREs in the version search.

-showversion
Prints product version and continues.

-verbose:<*option*>**[,**<*option*>**...]**
Enables verbose output. Separate multiple options using commas. The available options are:

class
Writes an entry to stderr for each class that is loaded.

gc See "Using verbose:gc information" on page 51.

jni
Writes information to stderr describing the JNI services called by the application and JVM.

sizes
Writes information to stderr describing the active memory usage settings.

stack
Writes information to stderr describing the Java and C stack usage for each thread.

-version
Prints out the version of the standard JVM. When used in conjunction with the **-Xrealtime** option, it prints out the Websphere Real Time product version.

-version:<*value*>
Requires the specified version to run, for example "1.5".

-X Prints help on nonstandard options.

Globalization of the java command

The java and javaw launchers accept arguments and class names containing any character that is in the character set of the current locale. You can also specify any Unicode character in the class name and arguments by using Java escape sequences.

To do this, use the **-Xargencoding** command-line option.

-Xargencoding
Use argument encoding. To specify a Unicode character, use escape sequences in the form `\u####`, where # is a hexadecimal digit (0 to 9, A to F).

-Xargencoding:utf8
Use UTF8 encoding.

-Xargencoding:latin
Use ISO8859_1 encoding.

For example, to specify a class called HelloWorld using Unicode encoding for both capital letters, use this command:

```
java -Xargencoding '\u0048ello\u0057orld'
```

The java and javaw commands provide translated messages. These messages differ based on the locale in which Java is running. The detailed error descriptions and other debug information that is returned by java is in English.

Specifying garbage collection policy for non Real-Time

The Garbage Collector manages the memory used by Java and by applications running in the JVM. These Garbage Collection policies are not available when the **-Xrealtime** option is specified.

When the Garbage Collector receives a request for storage, unused memory in the heap is set aside in a process called "allocation". The Garbage Collector also checks for areas of memory that are no longer referenced, and releases them for reuse. This is known as "collection".

The collection phase can be triggered by a memory allocation fault, which occurs when no space is left for a storage request, or by an explicit System.gc() call.

Garbage collection can significantly affect application performance, so the IBM virtual machine provides various methods of optimizing the way garbage collection is carried out, potentially reducing the effect on your application.

For more detailed information about garbage collection, see the Diagnostics Guide.

Garbage collection options

The **-Xgcpolicy** options control the behavior of the Garbage Collector. They make trade-offs between throughput of the application and overall system, and the pause times that are caused by garbage collection.

For definitions of **-Xgcpolicy:[optthruput]|[optavgpause]|[gencon]** see "Garbage collection options used with the non-real-time mode" on page 337.

Pause time

When an application's attempt to create an object cannot be satisfied immediately from the available space in the heap, the Garbage Collector is responsible for identifying unreferenced objects (garbage), deleting them, and returning the heap to a state in which the immediate and subsequent allocation requests can be satisfied quickly.

Such garbage collection cycles introduce occasional unexpected pauses in the execution of application code. Because applications grow in size and complexity, and heaps become correspondingly larger, this garbage collection pause time tends to grow in size and significance.

The default garbage collection value, **-Xgcpolicy:optthruput**, delivers very high throughput to applications, but at the cost of these occasional pauses, which can vary from a few milliseconds to many seconds, depending on the size of the heap and the quantity of garbage.

Pause time reduction

The JVM uses two techniques to reduce pause times: concurrent garbage collection and generational garbage collection.

The **-Xgcpolicy:optavgpause** command-line option requests the use of concurrent garbage collection to reduce significantly the time that is spent in garbage collection pauses. Concurrent GC reduces the pause time by performing some garbage collection activities concurrently with normal program execution to minimize the disruption caused by the collection of the heap. The **-Xgcpolicy:optavgpause** option also limits the effect of increasing the heap size on the length of the garbage collection pause. The **-Xgcpolicy:optavgpause** option is most useful for configurations that have large heaps. With the reduced pause time, you might experience some reduction of throughput to your applications.

During concurrent garbage collection, a significant amount of time is wasted identifying relatively long-lasting objects that cannot then be collected. If garbage collection concentrates on only the objects that are most likely to be recyclable, you

can further reduce pause times for some applications. Generational GC reduces pause times by dividing the heap into two generations: the “new” and the “tenure” areas. Objects are placed in one of these areas depending on their age. The new area is the smaller of the two and contains new objects; the tenure is larger and contains older objects. Objects are first allocated to the new area; if they have active references for long enough, they are promoted to the tenure area.

Generational GC depends on most objects not lasting long. Generational GC reduces pause times by concentrating the effort to reclaim storage on the new area because it has the most recyclable space. Rather than occasional but lengthy pause times to collect the entire heap, the new area is collected more frequently and, if the new area is small enough, pause times are comparatively short. However, generational GC has the drawback that, over time, the tenure area might become full. To minimize the pause time when this situation occurs, use a combination of concurrent GC and generational GC. The **-Xgcpolicy:gencon** option requests the combined use of concurrent and generational GC to help minimize the time that is spent in any garbage collection pause.

Environments with very full heaps

If the Java heap becomes nearly full, and very little garbage can be reclaimed, requests for new objects might not be satisfied quickly because no space is immediately available.

If the heap is operated at near-full capacity, application performance might suffer regardless of which garbage collection options are used; and, if requests for more heap space continue to be made, the application might receive an OutOfMemoryError, which results in JVM termination if the exception is not caught and handled. At this point, the JVM produces a Javadump file for use during diagnostics. In these conditions, you are recommended either to increase the heap size by using the **-Xmx** option or to reduce the number of objects in use.

For more information, see the Diagnostics Guide.

Euro symbol support

The IBM SDK and Runtime Environment set the Euro as the default currency for those countries in the European Monetary Union (EMU) for dates on or after 1 January, 2002. From 1 January 2008, Cyprus and Malta also have the Euro as the default currency.

To use the old national currency, specify **–Duser.variant=PREEURO** on the Java command line.

If you are running the UK, Danish, or Swedish locales and want to use the Euro, specify **–Duser.variant=EURO** on the Java command line.

Fallback font configuration files

The Linux fallback font configuration files (`fontconfig.RedHat.bfc` and `fontconfig.SuSE.bfc`) are installed to provide font settings suitable for new enterprise Linux distributions.

These files are for your convenience only. Their presence does not imply that the new Linux distribution is a supported platform for the IBM SDK and Runtime Environment for Linux platforms, Java Technology Edition, Version 6.

Using Indian and Thai input methods

From Version 6, the Indian and Thai input methods are not available by default. You must manually include the input method `jar` files in your Java extensions path to use the Indian and Thai input methods.

About this task

In Version 5.0, the input method `jar` files were included in the `jre/lib/ext` directory and were automatically loaded by the JVM. In Version 6, the input method `jar` files are included in the `jre/lib/im` directory and you must manually add them to the Java extensions path to enable Indian and Thai input methods.

Procedure

- Copy the `indicim.jar` and `thaiim.jar` files from the `jre/lib/im` directory to the `jre/lib/ext` directory.
- Add the `jre/lib/im` directory to the extension directories system property. Use the following command-line option:

```
java -Djava.ext.dirs=/opt/ibm/ibm-wrt-i386-60/jre/lib/ext:/opt/ibm/ibm-wrt-i386-60/jre/lib/im <class>
```

What to do next

If you installed the SDK or Runtime Environment in a different directory, replace /opt/ibm/ibm-wrt-i386-60/ with the directory in which you installed the SDK or Runtime Environment.

Developing Java applications

The SDK for Linux contains many tools and libraries required for Java software development.

See "Contents of the SDK" on page 359 for details of the tools available.

Using XML

The IBM SDK contains the XML4J and XL XP-J parsers, the XL TXE-J 1.0 XSLT compiler, and the XSLT4J XSLT interpreter. These tools allow you to parse, validate, transform, and serialize XML documents independently from any given XML processing implementation.

Use factory finders to locate implementations of the abstract factory classes, as described in "Selecting an XML processor" on page 373. By using factory finders, you can select a different XML library without changing your Java code.

Available XML libraries

The IBM SDK for Java contains the following XML libraries:

XML4J 4.5

XML4J is a validating parser providing support for the following standards:

- XML 1.0 (4th edition)
- Namespaces in XML 1.0 (2nd edition)
- XML 1.1 (2nd edition)
- Namespaces in XML 1.1 (2nd edition)

- W3C XML Schema 1.0 (2nd Edition)
- XInclude 1.0 (2nd Edition)
- OASIS XML Catalogs 1.0
- SAX 2.0.2
- DOM Level 3 Core, Load and Save
- DOM Level 2 Core, Events, Traversal and Range
- JAXP 1.4

XML4J 4.5 is based on Apache Xerces-J 2.9.0. See http://xerces.apache.org/xerces2-j/ for more information.

XL XP-J 1.1

XL XP-J 1.1 is a high-performance non-validating parser that provides support for StAX 1.0 (JSR 173). StAX is a bidirectional API for pull-parsing and streaming serialization of XML 1.0 and XML 1.1 documents. See the "XL XP-J reference information" on page 376 section for more details about what is supported by XL XP-J 1.1.

XL TXE-J 1.0.1 Beta

For Version 5.0, the IBM SDK for Java included the XSLT4J compiler and interpreter. The XSLT4J interpreter was used by default.

For Version 6, the IBM SDK for Java includes XL TXE-J. XL TXE-J includes the XSLT4J 2.7.8 interpreter and a new XSLT compiler. The new compiler is used by default. The XSLT4J compiler is no longer included with the IBM SDK for Java. See "Migrating to the XL-TXE-J" on page 374 for information about migrating to XL TXE-J.

XL TXE-J provides support for the following standards:

- XSLT 1.0
- XPath 1.0
- JAXP 1.4

Selecting an XML processor

XML processor selection is performed using service providers. When using a factory finder, Java looks in the following places, in this order, to see which service provider to use:

1. The system property with the same name as the service provider.
2. The service provider specified in a properties file.
   - **For XMLEventFactory, XMLInputFactory, and XMLOutputFactory only.** The value of the service provider in the file /opt/ibm/ibm-wrt-i386-60/jre/lib/stax.properties.
   - **For other factories.** The value of the service provider in the file /opt/ibm/ibm-wrt-i386-60/jre/lib/jaxp.properties.
3. The contents of the META-INF/services/<service.provider> file.
4. The default service provider.

The following service providers control the XML processing libraries used by Java:

javax.xml.parsers.SAXParserFactory
Selects the SAX parser. By default, org.apache.xerces.jaxp.SAXParserFactoryImpl from the XML4J library is used.

javax.xml.parsers.DocumentBuilderFactory
Selects the document builder. By default, org.apache.xerces.jaxp.DocumentBuilderFactoryImpl from the XML4J library is used.

javax.xml.datatype.DatatypeFactory
Selects the datatype factory. By default, org.apache.xerces.jaxp.datatype.DatatypeFactoryImpl from the XML4J library is used.

javax.xml.stream.XMLEventFactory
Selects the StAX event factory. By default, com.ibm.xml.xlxp.api.stax.XMLEventFactoryImpl from the XL XP-J library is used.

javax.xml.stream.XMLInputFactory
Selects the StAX parser. By default, com.ibm.xml.xlxp.api.stax.XMLInputFactoryImpl from the XL XP-J library is used.

javax.xml.stream.XMLOutputFactory
Selects the StAX serializer. By default, com.ibm.xml.xlxp.api.stax.XMLOutputFactoryImpl from the XL XP-J library is used.

javax.xml.transform.TransformerFactory
Selects the XSLT processor. Possible values are:

com.ibm.xtq.xslt.jaxp.compiler.TransformerFactoryImpl
Use the XL TXE-J compiler. This value is the default.

org.apache.xalan.processor.TransformerFactoryImpl
Use the XSLT4J interpreter.

javax.xml.validation.SchemaFactory:http://www.w3.org/2001/XMLSchema
Selects the schema factory for the W3C XML Schema language. By default, org.apache.xerces.jaxp.validation.XMLSchemaFactory from the XML4J library is used.

javax.xml.xpath.XPathFactory
Selects the XPath processor. By default, org.apache.xpath.jaxp.XPathFactoryImpl from the XSLT4J library is used.

Migrating to the XL-TXE-J

The XL TXE-J compiler has replaced the XSLT4J interpreter as the default XSLT processor. Follow these steps to prepare your application for the new library.

About this task

The XL TXE-J compiler is faster than the XSLT4J interpreter when you are applying the same transformation more than once. If you perform each individual transformation only once, the XL TXE-J compiler is slower than the XSLT4J interpreter because compilation and optimization reduce performance.

To continue using the XSLT4J interpreter as your XSLT processor, set the **javax.xml.transform.TransformerFactory** service provider to org.apache.xalan.processor.TransformerFactoryImpl.

To migrate to the XL-TXE-J compiler, follow the instructions in this task.

Procedure

1. Use com.ibm.xtq.xslt.jaxp.compiler.TransformerFactoryImpl when setting the **javax.xml.transform.TransformerFactory** service provider.
2. Regenerate class files generated by the XSLT4J compiler. XL TXE-J cannot execute class files generated by the XSLT4J compiler.
3. Some methods generated by the compiler might exceed the JVM method size limit, in which case the compiler attempts to split these methods into smaller methods.
   - If the compiler splits the method successfully, you receive the following warning:

     `Some generated functions exceeded the JVM method size limit and were automatically split into smaller functions. You might get better performance by manually splitting very large templates into smaller templates, by using the 'splitlimit' option to the Process or Compile command, or by setting the 'http://www.ibm.com/xmlns/prod/xltxe-j/split-limit' transformer factory attribute.`You can use the compiled classes, but you might get better performance by controlling the split limit manually.
   - If the compiler does not split the method successfully, you receive one of the following exceptions:

     `com.ibm.xtq.bcel.generic.ClassGenException: Branch target offset too large for short` or

     `bytecode array size > 65535 at offset=#####`Try setting the split limit manually, or using a lower split limit.

   To set the split limit, use the **-SPLITLIMIT** option when using the Process or Compile commands, or the **http://www.ibm.com/xmlns/prod/xltxe-j/split-limit** transformer factory attribute when using the transformer factory. The split limit can be between 100 and 2000. When setting the split limit manually, use the highest split limit possible for best performance.
4. XL TXE-J might need more memory than the XSLT4J compiler. If you are running out of memory or performance seems slow, increase the size of the heap using the **-Xmx** option.
5. Migrate your application to use the new attribute keys. The old transformer factory attribute keys are deprecated. The old names are accepted with a warning.

Table 21. Changes to attribute keys from the XSL4J compiler to the XL TXE-J compiler

XSL4J compiler attribute	XL TXE-J compiler attribute
translet-name	http://www.ibm.com/xmlns/prod/xltxe-j/translet-name
destination-directory	http://www.ibm.com/xmlns/prod/xltxe-j/destination-directory
package-name	http://www.ibm.com/xmlns/prod/xltxe-j/package-name
jar-name	http://www.ibm.com/xmlns/prod/xltxe-j/jar-name
generate-translet	http://www.ibm.com/xmlns/prod/xltxe-j/generate-translet
auto-translet	http://www.ibm.com/xmlns/prod/xltxe-j/auto-translet
use-classpath	http://www.ibm.com/xmlns/prod/xltxe-j/use-classpath
debug	http://www.ibm.com/xmlns/prod/xltxe-j/debug
indent-number	http://www.ibm.com/xmlns/prod/xltxe-j/indent-number
enable-inlining	*Obsolete in new compiler*

6. Optional: For best performance, ensure that you are not recompiling XSLT transformations that can be reused. Use one of the following methods to reuse compiled transformations:
   - If your stylesheet does not change at runtime, compile the stylesheet as part of your build process and put the compiled classes on your classpath. Use the org.apache.xalan.xsltc.cmdline.Compile command to compile the stylesheet and set the **http://www.ibm.com/xmlns/prod/xltxe-j/use-classpath** transformer factory attribute to `true` to load the classes from the classpath.
   - If your application will use the same stylesheet during multiple runs, set the **http://www.ibm.com/xmlns/prod/xltxe-j/auto-translet** transformer factory attribute to `true` to automatically save the compiled stylesheet to disk for reuse. The compiler will use a compiled stylesheet if it is available, and compile the stylesheet if it is not available or is out-of-date. Use the **http://www.ibm.com/xmlns/prod/xltxe-j/destination-directory** transformer factory attribute to set the directory used to store compiled stylesheets. By default, compiled stylesheets are stored in the same directory as the stylesheet.
   - If your application is a long-running application that reuses the same stylesheet, use the transformer factory to compile the stylesheet and create a Templates object. You can use the Templates object to create Transformer objects without recompiling the stylesheet. The Transformer objects can also be reused but are not thread-safe.

XML reference information

The XL XP-J and XL TXE-J XML libraries are new for Version 6 of the SDK. This reference information describes the features supported by these libraries.

XL XP-J reference information:

XL XP-J 1.1 is a high-performance non-validating parser that provides support for StAX 1.0 (JSR 173). StAX is a bidirectional API for pull-parsing and streaming serialization of XML 1.0 and XML 1.1 documents.

Unsupported features

The following optional StAX features are not supported by XL XP-J:

- DTD validation when using an XMLStreamReader or XMLEventReader. The XL XP-J parser is non-validating.
- When using an XMLStreamReader to read from a character stream (java.io.Reader), the Location.getCharaterOffset() method always returns -1. The Location.getCharaterOffset() returns the byte offset of a Location when using an XMLStreamReader to read from a byte stream (java.io.InputStream).

XMLInputFactory reference

The javax.xml.stream.XMLInputFactory implementation supports the following properties, as described in the XMLInputFactory Javadoc information: http://java.sun.com/javase/6/docs/api/javax/xml/stream/XMLInputFactory.html.

Property name	Supported?
javax.xml.stream.isValidating	No. The XL XP-J scanner does not support validation.

Property name	Supported?
javax.xml.stream.isNamespaceAware	Yes, supports true and false. For XMLStreamReaders created from DOMSources, namespace processing depends on the methods that were used to create the DOM tree, and this value has no effect.
javax.xml.stream.isCoalescing	Yes
javax.xml.stream.isReplacingEntityReferences	Yes. For XMLStreamReaders created from DOMSources, if entities have already been replaced in the DOM tree, setting this parameter has no effect.
javax.xml.stream.isSupportingExternalEntities	Yes
javax.xml.stream.supportDTD	True is always supported. Setting the value to false works only if the **com.ibm.xml.xlxp.support.dtd.compat.mode** system property is also set to false. When both properties are set to false, parsers created by the factory throw an XMLStreamException when they encounter an entity reference that requires expansion. This setting is useful for protecting against Denial of Service (DoS) attacks involving entities declared in the DTD. Setting the value to false does not work before Service Refresh 2.
javax.xml.stream.reporter	Yes
javax.xml.stream.resolver	Yes

XL XP-J also supports the optional method createXMLStreamReader(javax.xml.transform.Source), which allows StAX readers to be created from DOM and SAX sources.

XL XP-J also supports the **javax.xml.stream.isSupportingLocationCoordinates** property. If you set this property to true, XMLStreamReaders created by the factory return accurate line, column, and character information using Location objects. If you set this property to false, line, column, and character information is not available. By default, this property is set to false for performance reasons.

XMLStreamReader reference

The javax.xml.stream.XMLStreamReader implementation supports the following properties, as described in the XMLStreamReader Javadoc: http://java.sun.com/javase/6/docs/api/javax/xml/stream/XMLStreamReader.html.

Property name	Supported?
javax.xml.stream.entities	Yes
javax.xml.stream.notations	Yes

XL XP-J also supports the **javax.xml.stream.isInterning** property. This property returns a boolean value indicating whether or not XML names and namespace

URIs returned by the API calls have been interned by the parser. This property is read-only.

XMLOutputFactory reference

The javax.xml.stream.XMLOutputFactory implementation supports the following properties, as described in the XMLOutputFactory Javadoc: http://java.sun.com/javase/6/docs/api/javax/xml/stream/XMLOutputFactory.html.

Property name	Supported?
javax.xml.stream.isRepairingNamespaces	Yes

XL XP-J also supports the **javax.xml.stream.XMLOutputFactory.recycleWritersOnEndDocument** property. If you set this property to true, XMLStreamWriters created by this factory are recycled when writeEndDocument() is called. After recycling, some XMLStreamWriter methods, such as getNamespaceContext(), must not be called. By default, XMLStreamWriters are recycled when close() is called. You must call the XMLStreamWriter.close() method when you have finished with an XMLStreamWriter, even if this property is set to true.

XMLStreamWriter reference

The javax.xml.stream.XMLStreamWriter implementation supports the following properties, as described in the XMLStreamWriter Javadoc: http://java.sun.com/javase/6/docs/api/javax/xml/stream/XMLStreamWriter.html.

Property name	Supported?
javax.xml.stream.isRepairingNamespaces	Yes

Properties on XMLStreamWriter objects are read-only.

XL XP-J also supports the **javax.xml.stream.XMLStreamWriter.isSetPrefixBeforeStartElement** property. This property returns a Boolean indicating whether calls to setPrefix() and setDefaultNamespace() should occur before calls to writeStartElement() or writeEmptyElement() to put a namespace prefix in scope for that element. XL XP-J always returns false; calls to setPrefix() and setDefaultNamespace() should occur after writeStartElement() or writeEmptyElement().

XL TXE-J reference information:

XL TXE-J is an XSLT library containing the XSLT4J 2.7.8 interpreter and a XSLT compiler.

Feature comparison table

Table 22. Comparison of the features in the XSLT4J interpreter, the XSLT4J compiler, and the XL TXE-J compiler.

Feature	XSLT4J interpreter (included)	XSLT4J compiler (not included)	XL TXE-J compiler (included)
http://javax.xml.transform.stream.StreamSource/feature feature	Yes	Yes	Yes
http://javax.xml.transform.stream.StreamResult/feature feature	Yes	Yes	Yes
http://javax.xml.transform.dom.DOMSource/feature feature	Yes	Yes	Yes
http://javax.xml.transform.dom.DOMResult/feature feature	Yes	Yes	Yes
http://javax.xml.transform.sax.SAXSource/feature feature	Yes	Yes	Yes
http://javax.xml.transform.sax.SAXResult/feature feature	Yes	Yes	Yes
http://javax.xml.transform.stax.StAXSource/feature feature	Yes	No	Yes
http://javax.xml.transform.stax.StAXResult/feature feature	Yes	No	Yes
http://javax.xml.transform.sax.SAXTransformerFactory/feature feature	Yes	Yes	Yes
http://javax.xml.transform.sax.SAXTransformerFactory/feature/xmlfilter feature	Yes	Yes	Yes
http://javax.xml.XMLConstants/feature/secure-processing feature	Yes	Yes	Yes
http://xml.apache.org/xalan/features/incremental attribute	Yes	No	No
http://xml.apache.org/xalan/features/optimize attribute	Yes	No	No
http://xml.apache.org/xalan/properties/source-location attribute	Yes	No	No
translet-name attribute	N/A	Yes	Yes (with new name)
destination-directory attribute	N/A	Yes	Yes (with new name)
package-name attribute	N/A	Yes	Yes (with new name)

Table 22. Comparison of the features in the XSLT4J interpreter, the XSLT4J compiler, and the XL TXE-J compiler. (continued)

Feature	XSLT4J interpreter (included)	XSLT4J compiler (not included)	XL TXE-J compiler (included)
jar-name attribute	N/A	Yes	Yes (with new name)
generate-translet attribute	N/A	Yes	Yes (with new name)
auto-translet attribute	N/A	Yes	Yes (with new name)
use-classpath attribute	N/A	Yes	Yes (with new name)
enable-inlining attribute	No	Yes	No (obsolete in TL TXE-J)
indent-number attribute	No	Yes	Yes (with new name)
debug attribute	No	Yes	Yes (with new name)
Java extensions	Yes	Yes (abbreviated syntax only, xalan:component/xalan:script constructs not supported)	
JavaScript extensions	Yes	No	No
Extension elements	Yes	No	No
EXSLT extension functions	Yes	Yes (excluding dynamic)	Yes (excluding dynamic)
redirect extension	Yes	Yes (excluding redirect:open and redirect:close)	Yes
output extension	No	Yes	Yes
nodeset extension	Yes	Yes	Yes
NodeInfo extension functions	Yes	No	No
SQL library extension	Yes	No	No
pipeDocument extension	Yes	No	No
evaluate extension	Yes	No	No
tokenize extension	Yes	No	No
XML 1.1	Yes	Yes	Yes

Notes

1. With the Process command, use **-FLAVOR sr2sw** to transform using StAX stream processing, and **-FLAVOR er2ew** for StAX event processing.
2. The new compiler does not look for the org.apache.xalan.xsltc.dom.XSLTCDTMManager service provider. Instead, if StreamSource is used, the compiler switches to a high-performance XML parser.
3. Inlining is obsolete in XL TXE-J.
   - The **-XN** option to the Process command is silently ignored.
   - The **-n** option to the Compile command is silently ignored.

- The **enable-inlining** transformer factory attribute is silently ignored.

4. The org.apache.xalan.xsltc.trax.SmartTransformerFactoryImpl class is no longer supported.

Using an older version of Xerces or Xalan:

If you are using an older version of Xerces (before 2.0) or Xalan (before 2.3) in the endorsed override, you might get a NullPointerException when you start your application. This exception occurs because these older versions do not handle the `jaxp.properties` file correctly.

About this task

To avoid this situation, use one of the following workarounds:

- Upgrade to a newer version of the application that implements the latest Java API for XML Programming (JAXP) specification (https://jaxp.dev.java.net/).
- Remove the `jaxp.properties` file from /opt/ibm/ibm-wrt-i386-60/jre/lib.
- Uncomment the entries in the `jaxp.properties` file in /opt/ibm/ibm-wrt-i386-60/jre/lib.
- Set the system property for **javax.xml.parsers.SAXParserFactory**, **javax.xml.parsers.DocumentBuilderFactory**, or **javax.xml.transform.TransformerFactory** using the **-D** command-line option.
- Set the system property for **javax.xml.parsers.SAXParserFactory**, **javax.xml.parsers.DocumentBuilderFactory**, or **javax.xml.transform.TransformerFactory** in your application. For an example, see the JAXP 1.4 specification.
- Explicitly set the SAX parser, Document builder, or Transformer factory using the **IBM_JAVA_OPTIONS** environment variable.

```
export IBM_JAVA_OPTIONS=-Djavax.xml.parsers.SAXParserFactory=
  org.apache.xerces.jaxp.SAXParserFactoryImpl
```

or

```
export IBM_JAVA_OPTIONS=-Djavax.xml.parsers.DocumentBuilderFactory=
  org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
```

or

```
export IBM_JAVA_OPTIONS=-Djavax.xml.transform.TransformerFactory=
  org.apache.xalan.processor.TransformerFactoryImpl
```

Debugging Java applications

To debug Java programs, you can use the Java Debugger (JDB) application or other debuggers that communicate by using the Java Platform Debugger Architecture (JPDA) that is provided by the SDK for the operating system.

More information about problem diagnosis using Java can be found in the Diagnostics Guide.

Java Debugger (JDB)

The Java Debugger (JDB) is included in the SDK for Linux. The debugger is started with the jdb command; it attaches to the JVM using JPDA.

To debug a Java application:

1. Start the JVM with the following options:

```
java -Xdebug -Xrunjdwp:transport=dt_socket,server=y,address=<port> <class>
```

The JVM starts up, but suspends execution before it starts the Java application.

2. In a separate session, you can attach the debugger to the JVM:

```
jdb -attach <port>
```

The debugger will attach to the JVM, and you can now issue a range of commands to examine and control the Java application; for example, type `run` to allow the Java application to start.

For more information about JDB options, type:

```
jdb -help
```

For more information about JDB commands:

1. Type `jdb`
2. At the jdb prompt, type `help`

You can also use JDB to debug Java applications running on remote workstations. JPDA uses a TCP/IP socket to connect to the remote JVM.

1. Start the JVM with the following options:

```
java -Xdebug -Xrunjdwp:transport=dt_socket,server=y,address=<port> <class>
```

The JVM starts up, but suspends execution before it starts the Java application.

2. Attach the debugger to the remote JVM:

```
jdb -attach <host>:<port>
```

The Java Virtual Machine Debugging Interface (JVMDI) is not supported in this release. It has been replaced by the Java Virtual Machine Tool Interface (JVMTI).

For more information about JDB and JPDA and their usage, see these Web sites:

- http://java.sun.com/products/jpda/
- http://java.sun.com/javase/6/docs/technotes/guides/jpda/
- http://java.sun.com/javase/6/docs/technotes/guides/jpda/jdb.html

Selective debugging

Use the com.ibm.jvm.Debuggable annotation to mark classes and methods that should be available for debugging. Use the **-XselectiveDebug** parameter to enable selective debugging at run time. The JVM optimizes methods that do not need debugging to provide better performance in a debugging environment.

About this task

Selective debugging is useful when Java is being used as a framework for development, for example, as an IDE. The Java code for the IDE is optimized for performance while the user code is debugged.

Procedure

1. Import the Debuggable annotation from the com.ibm.jvm package.

```
import com.ibm.jvm.Debuggable;
```

2. Decorate methods using the Debuggable annotation.

```
@Debuggable
public int method1() {
  ...
}
```

3. Optional: You can also decorate classes using the Debuggable annotation. All methods in the class will remain debuggable.

```
@Debuggable
public class Class1 {
  ...
}
```

4. Enable selective debugging at run time using the **-XselectiveDebug** command-line option.

Results

Applications will run faster while being debugged because the core Java API and any IDE code can be optimized for performance.

Determining whether your application is running on a 32-bit or 64-bit JVM

Some Java applications must be able to determine whether they are running on a 32-bit JVM or on a 64-bit JVM. For example, if your application has a native code library, the library must be compiled separately in 32- and 64-bit forms for platforms that support both 32- and 64-bit modes of operation. In this case, your application must load the correct library at runtime, because it is not possible to mix 32- and 64-bit code.

About this task

The system property **com.ibm.vm.bitmode** allows applications to determine the mode in which your JVM is running. It returns the following values:

- 32 - the JVM is running in 32-bit mode (31-bit mode for Linux on System z)
- 64 - the JVM is running in 64-bit mode

You can inspect the **com.ibm.vm.bitmode** property from inside your application code using the call:

```
System.getProperty("com.ibm.vm.bitmode");
```

How the JVM processes signals

When a signal is raised that is of interest to the JVM, a signal handler is called. This signal handler determines whether it has been called for a Java or non-Java thread.

If the signal is for a Java thread, the JVM takes control of the signal handling. If an application handler for this signal is installed and you did not specify the **-Xnosigchain** command-line option, the application handler for this signal is called after the JVM has finished processing.

If the signal is for a non-Java thread, and the application that installed the JVM had previously installed its own handler for the signal, control is given to that handler. Otherwise, if the signal is requested by the JVM or Java application, the signal is ignored or the default action is taken.

For exception and error signals, the JVM either:

- Handles the condition and recovers, or

- Enters a controlled shut down sequence where it:
  1. Produces dumps, to describe the JVM state at the point of failure
  2. Calls your application's signal handler for that signal
  3. Calls any application-installed unexpected shut down hook
  4. Performs the necessary JVM cleanup

For information about writing a launcher that specifies the above hooks, see: http://www.ibm.com/developerworks/java/library/i-signalhandling/. This item was written for Java V1.3.1, but still applies to later versions.

For interrupt signals, the JVM also enters a controlled shut down sequence, but this time it is treated as a normal termination that:

1. Calls your application's signal handler for that signal
2. Calls all application shut down hooks
3. Calls any application-installed exit hook
4. Performs the necessary JVM cleanup

The shut down is identical to the shut down initiated by a call to the Java method `System.exit()`.

Other signals that are used by the JVM are for internal control purposes and do not cause it to stop. The only control signal of interest is SIGQUIT, which causes a Javadump to be generated.

Signals used by the JVM

The types of signals are Exceptions, Errors, Interrupts, and Controls.

Table 23 shows the signals that are used by the JVM. The signals are grouped in the table by type or use, as follows:

Exceptions
: The operating system synchronously raises an appropriate exception signal whenever an unrecoverable condition occurs.

Errors The JVM raises a SIGABRT if it detects a condition from which it cannot recover.

Interrupts
: Interrupt signals are raised asynchronously, from outside a JVM process, to request shut down.

Controls
: Other signals that are used by the JVM for control purposes.

Table 23. Signals used by the JVM

Signal Name	Signal type	Description	Disabled by -Xrs
SIGBUS (7)	Exception	Incorrect access to memory (data misalignment)	Yes
SIGSEGV (11)	Exception	Incorrect access to memory (write to inaccessible memory)	Yes
SIGILL (4)	Exception	Illegal instruction (attempt to call an unknown machine instruction)	No

Table 23. Signals used by the JVM (continued)

Signal Name	Signal type	Description	Disabled by -Xrs
SIGFPE (8)	Exception	Floating point exception (divide by zero)	Yes
SIGABRT (6)	Error	Abnormal termination. The JVM raises this signal whenever it detects a JVM fault.	Yes
SIGINT (2)	Interrupt	Interactive attention (CTRL-C). JVM exits normally.	Yes
SIGTERM (15)	Interrupt	Termination request. JVM will exit normally.	Yes
SIGHUP (1)	Interrupt	Hang up. JVM exits normally.	Yes
SIGQUIT (3)	Control	A quit signal for a terminal. By default, this triggers a Javadump.	Yes
SIGTRAP (5)	Control	Used by the JIT.	Yes
SIGRTMIN (32)	Control	Used by the JVM for internal control purposes.	No
SIGRTMAX (2)	Control	Used by the SDK.	No
SIGCHLD (17)	Control	Used by the SDK for internal control.	No

Note: A number supplied after the signal name is the standard numeric value for that signal.

Use the **-Xrs** (reduce signal usage) option to prevent the JVM from handling most signals. For more information, see Sun's Java application launcher page.

Signals 1 (SIGHUP), 2 (SIGINT), 4 (SIGILL), 7 (SIGBUS), 8 (SIGFPE), 11 (SIGSEGV), and 15 (SIGTERM) on JVM threads cause the JVM to shut down; therefore, an application signal handler should not attempt to recover from these unless it no longer requires the JVM.

Linking a native code driver to the signal-chaining library

The Runtime Environment contains signal-chaining. Signal-chaining enables the JVM to interoperate more efficiently with native code that installs its own signal handlers.

About this task

Signal-chaining enables an application to link and load the shared library `libjsig.so` before the system libraries. The libjsig.so library ensures that calls such as signal(), sigset(), and sigaction() are intercepted so that their handlers do not replace the JVM's signal handlers. Instead, these calls save the new signal handlers,

or "chain" them behind the handlers that are installed by the JVM. Later, when any of these signals are raised and found not to be targeted at the JVM, the preinstalled handlers are invoked.

If you install signal handlers that use sigaction() , some **sa_flags** are not observed when the JVM uses the signal. These are:

- SA_NOCLDSTOP - This is always unset.
- SA_NOCLDWAIT - This is always unset.
- SA_RESTART - This is always set.

The libjsig.so library also hides JVM signal handlers from the application. Therefore, calls such as signal(), sigset(), and sigaction() that are made after the JVM has started no longer return a reference to the JVM's signal handler, but instead return any handler that was installed before JVM startup.

To use libjsig.so:

- Link it with the application that creates or embeds a JVM:

  ```
  gcc -L$JAVA_HOME/bin -ljsig -L$JAVA_HOME/bin/j9vm -ljvm java_application.c
  ```

  or
- Use the **LD_PRELOAD** environment variable:

  ```
  export LD_PRELOAD=$JAVA_HOME/bin/libjsig.so; java_application (bash and ksh)

  setenv LD_PRELOAD=$JAVA_HOME/bin/libjsig.so; java_application (csh)
  ```

The environment variable **JAVA_HOME** should be set to the location of the SDK, for example,/opt/ibm/ibm-wrt-i386-60/.

To use libjsig.a:

- Link it with the application that creates or embeds a JVM:

  ```
  cc_r  <other compile/link parameter> -L/opt/ibm/ibm-wrt-i386-60/jre/bin -ljsig
  -L/opt/ibm/ibm-wrt-i386-60/jre/bin/j9vm -ljvm java_application.c
  ```

  Note: Use `xlc_r` or `xlC_r` in place of `cc_r` if that is how you usually call the compiler or linker.

Writing JNI applications

Valid Java Native Interface (JNI) version numbers that programs can specify on the JNI_CreateJavaVM() API call are: JNI_VERSION_1_2(0x00010002) and JNI_VERSION_1_4(0x00010004).

Restriction: Version 1.1 of the JNI is not supported.

This version number determines only the level of the JNI to use. The actual level of the JVM that is created is specified by the JSE libraries (that is, v6). The JNI level *does not* affect the language specification that is implemented by the JVM, the class library APIs, or any other area of JVM behavior. For more information, see http://java.sun.com/javase/6/docs/technotes/guides/jni/.

If your application needs two JNI libraries, one built for 32- and the other for 64-bit, use the **com.ibm.vm.bitmode** system property to determine if you are running with a 32- or 64-bit JVM and choose the appropriate library.

To compile and link a local application with the SDK, use the following command:

```
gcc -I/opt/ibm/ibm-wrt-i386-60/include -L/opt/ibm/ibm-wrt-i386-60/jre/lib/<arch>/j9vm
-ljvm -ldl -lpthread <JNI program filename>
```

The **-ljvm** option specifies that `libjvm.so` is the shared library that implements the JVM. The **-lpthread** option indicates that you are using native pthread support; if you do not link with the pthread library, a segmentation fault (signal SIGSEGV) might be caused when you run the JNI program.

Support for thread-level recovery of blocked connectors

Four new IBM-specific SDK classes have been added to the com.ibm.jvm package to support the thread-level recovery of Blocked connectors. The new classes are packaged in `core.jar`.

These classes allow you to unblock threads that have become blocked on networking or synchronization calls. If an application does not use these classes, it must end the whole process, rather than interrupting an individual blocked thread.

The classes are:

public interface InterruptibleContext
Defines two methods, isBlocked() and unblock(). The other three classes implement InterruptibleContext.

public class InterruptibleLockContext
A utility class for interrupting synchronization calls.

public class InterruptibleIOContext
A utility class for interrupting network calls.

public class InterruptibleThread
A utility class that extends java.lang.Thread, to allow wrapping of interruptible methods. It uses instances of InterruptibleLockContext and InterruptibleIOContext to perform the required isBlocked() and unblock() methods depending on whether a synchronization or networking operation is blocking the thread.

Both InterruptibleLockContext and InterruptibleIOContext work by referencing the current thread. Therefore if you do not use InterruptibleThread, you must provide your own class that extends java.lang.Thread, to use these new classes.

The Javadoc information for these classes is provided with the SDK in the `docs/content/apidoc` directory.

Configuring large page memory allocation

You can enable large page support, on systems that support it, by starting Java with the **-Xlp** option.

About this task

Large page usage is primarily intended to provide performance improvements to applications that allocate a great deal of memory and frequently access that memory. The large page performance improvements are a result of the reduced number of misses in the Translation Lookaside Buffer (TLB). The TLB maps a larger virtual storage area range and thus causes this improvement.

Large page support must be available in the kernel, and enabled, so that Java can use large pages.

To configure large page memory allocation, first ensure that the running kernel supports large pages. Check that the file `/proc/meminfo` contains the following lines:

```
HugePages_Total:     <number of pages>
HugePages_Free:      <number of pages>
Hugepagesize:        <page size, in kB>
```

The number of pages available and their sizes vary between distributions.

If large page support is not available in your kernel, these lines are not present in the `/proc/meminfo` file. In this case, you must install a new kernel containing support for large pages.

If large page support is available, but not enabled, `HugePages_Total` has the value 0. In this case, your administrator must enable large page support. Check your operating system manual for more instructions.

For the JVM to use large pages, your system must have an adequate number of contiguous large pages available. If large pages cannot be allocated, even when enough pages are available, possibly the large pages are not contiguous. Configuring the number of large pages at boot up creates them contiguously.

Large page allocations only succeed if the user is a member of the group with its gid stored in `/proc/sys/vm/hugetlb_shm_group`, or if Java is run with root access. See http://devresources.linux-foundation.org/dev/robustmutexes/src/fusyn.hg/Documentation/vm/hugetlbpage.txt for more information.

If a non-root user needs access to large pages, their locked memory limit must be increased. The locked memory limit must be at least as large as the maximum size of the Java heap. The maximum size of the Java heap can be specified using the **-Xmx** command-line option, or determined by adding **-verbose:sizes** and inspecting the output for the value `-Xmx`. If pam is not installed, change the locked memory limit using the ulimit command. If pam is installed, change the locked memory limit by adding the following lines to `/etc/security/limits.conf`:

```
@<large group name> soft memlock 2097152
@<large group name> hard memlock 2097152
```

Where *<large group name>* is the name of the group with its gid stored in `/proc/sys/vm/hugetlb_shm_group`.

CORBA support

The Java Platform, Standard Edition (JSE) supports, at a minimum, the specifications that are defined in the compliance document from Sun. In some cases, the IBM JSE ORB supports more recent versions of the specifications.

The minimum specifications supported are defined in the Official Specifications for CORBA support in Java SE 6: http://java.sun.com/javase/6/docs/api/org/omg/CORBA/doc-files/compliance.html.

Support for GIOP 1.2

This SDK supports all versions of GIOP, as defined by chapters 13 and 15 of the CORBA 2.3.1 specification, OMG document *formal/99-10-07*.

http://www.omg.org/cgi-bin/doc?formal/99-10-07

Bidirectional GIOP is not supported.

Support for Portable Interceptors

This SDK supports Portable Interceptors, as defined by the OMG in the document *ptc/01–03–04*, which you can obtain from:

http://www.omg.org/cgi-bin/doc?ptc/01–03-04

Portable Interceptors are hooks into the ORB that ORB services can use to intercept the normal flow of execution of the ORB.

Support for Interoperable Naming Service

This SDK supports the Interoperable Naming Service, as defined by the OMG in the document *ptc/00-08-07*, which you can obtain from:

http://www.omg.org/cgi-bin/doc?ptc/00-08-07

The default port that is used by the Transient Name Server (the tnameserv command), when no **ORBInitialPort** parameter is given, has changed from *900* to *2809*, which is the port number that is registered with the IANA (Internet Assigned Number Authority) for a CORBA Naming Service. Programs that depend on this default might have to be updated to work with this version.

The initial context that is returned from the Transient Name Server is now an org.omg.CosNaming.NamingContextExt. Existing programs that narrow the reference to a context org.omg.CosNaming.NamingContext still work, and do not need to be recompiled.

The ORB supports the **-ORBInitRef** and **-ORBDefaultInitRef** parameters that are defined by the Interoperable Naming Service specification, and the ORB::string_to_object operation now supports the ObjectURL string formats (corbaloc: and corbaname:) that are defined by the Interoperable Naming Service specification.

The OMG specifies a method ORB::register_initial_reference to register a service with the Interoperable Naming Service. However, this method is not available in the Sun Java Core API at Version 6. Programs that have to register a service in the current version must invoke this method on the IBM internal ORB implementation class. For example, to register a service “MyService”:

```
((com.ibm.CORBA.iiop.ORB)orb).register_initial_reference("MyService",
serviceRef);
```

Where `orb` is an instance of org.omg.CORBA.ORB, which is returned from ORB.init(), and `serviceRef` is a CORBA Object, which is connected to the ORB. This mechanism is an interim one, and is not compatible with future versions or portable to non-IBM ORBs.

System properties for tracing the ORB

A runtime debug feature provides improved serviceability. You might find it useful for problem diagnosis or it might be requested by IBM service personnel.

Tracing Properties

com.ibm.CORBA.Debug=*true*
Turns on ORB tracing.

com.ibm.CORBA.CommTrace=*true*
Adds GIOP messages (sent and received) to the trace.

com.ibm.CORBA.Debug.Output=<*file*>
Specify the trace output file. By default, this is of the form `orbtrc.DDMMYYYY.HHmm.SS.txt`.

Example of ORB tracing

For example, to trace events and formatted GIOP messages from the command line, type:

```
java -Dcom.ibm.CORBA.Debug=true
     -Dcom.ibm.CORBA.CommTrace=true <myapp>
```

Limitations

Do not enable tracing for normal operation, because it might cause performance degradation. Even if you have switched off tracing, FFDC (First Failure Data Capture) is still working, so serious errors are reported. If a debug output file is generated, examine it to check on the problem. For example, the server might have stopped without performing an ORB.shutdown().

The content and format of the trace output might vary from version to version.

System properties for tuning the ORB

The ORB can be tuned to work well with your specific network. The properties required to tune the ORB are described here.

com.ibm.CORBA.FragmentSize=<*size in bytes*>
Used to control GIOP 1.2 fragmentation. The default size is 1024 bytes.

To disable fragmentation, set the fragment size to 0 bytes:

```
java -Dcom.ibm.CORBA.FragmentSize=0 <myapp>
```

com.ibm.CORBA.RequestTimeout=<*time in seconds*>
Sets the maximum time to wait for a CORBA Request. By default the ORB waits indefinitely. Do not set the timeout too low to avoid connections ending unnecessarily.

com.ibm.CORBA.LocateRequestTimeout=<*time in seconds*>
Set the maximum time to wait for a CORBA LocateRequest. By default the ORB waits indefinitely.

com.ibm.CORBA.ListenerPort=<*port number*>
Set the port for the ORB to read incoming requests on. If this property is set, the ORB starts listening as soon as it is initialized. Otherwise, it starts listening only when required.

Java security permissions for the ORB

When running with a Java SecurityManager, invocation of some methods in the CORBA API classes might cause permission checks to be made, which might result in a SecurityException. If your program uses any of these methods, ensure that it is granted the necessary permissions.

Table 24. Methods affected when running with Java SecurityManager

Class/Interface	Method	Required permission
org.omg.CORBA.ORB	init	java.net.SocketPermission resolve

Table 24. Methods affected when running with Java SecurityManager (continued)

Class/Interface	Method	Required permission
org.omg.CORBA.ORB	connect	java.net.SocketPermission listen
org.omg.CORBA.ORB	resolve_initial_references	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_is_a	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_non_existent	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	OutputStream _request (String, boolean)	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_get_interface_def	java.net.SocketPermission connect
org.omg.CORBA. Request	invoke	java.net.SocketPermission connect
org.omg.CORBA. Request	send_deferred	java.net.SocketPermission connect
org.omg.CORBA. Request	send_oneway	java.net.SocketPermission connect
javax.rmi. PortableRemoteObject	narrow	java.net.SocketPermission connect

ORB implementation classes

A list of the ORB implementation classes.

The ORB implementation classes in this release are:

- org.omg.CORBA.ORBClass=com.ibm.CORBA.iiop.ORB
- org.omg.CORBA.ORBSingletonClass=com.ibm.rmi.corba.ORBSingleton
- javax.rmi.CORBA.UtilClass=com.ibm.CORBA.iiop.UtilDelegateImpl
- javax.rmi.CORBA.StubClass=com.ibm.rmi.javax.rmi.CORBA.StubDelegateImpl
- javax.rmi.CORBA.PortableRemoteObjectClass=com.ibm.rmi.javax.rmi.PortableRemoteObject

These are the default values, and you are advised not to set these properties or refer to the implementation classes directly. For portability, make references only to the CORBA API classes, and not to the implementation. These values might be changed in future releases.

RMI over IIOP

Java Remote Method Invocation (RMI) provides a simple mechanism for distributed Java programming. RMI over IIOP (RMI-IIOP) uses the Common Object Request Broker Architecture (CORBA) standard Internet Inter-ORB Protocol (IIOP) to extend the base Java RMI to perform communication. This allows direct interaction with any other CORBA Object Request Brokers (ORBs), whether they were implemented in Java or another programming language.

The following documentation is available:

- The RMI-IIOP Programmer's Guide is an introduction to writing RMI-IIOP programs.

- The *Java Language to IDL Mapping* document is a detailed technical specification of RMI-IIOP: http://www.omg.org/cgi-bin/doc?ptc/00-01-06.pdf.

Implementing the Connection Handler Pool for RMI

Thread pooling for RMI Connection Handlers is not enabled by default.

About this task

To enable the connection pooling implemented at the RMI TCPTransport level, set the option

```
-Dsun.rmi.transport.tcp.connectionPool=true
```

This version of the Runtime Environment does not have a setting that you can use to limit the number of threads in the connection pool.

Enhanced BigDecimal

From Java 5.0, the IBM BigDecimal class has been adopted by Sun as java.math.BigDecimal. The com.ibm.math.BigDecimal class is reserved for possible future use by IBM and is currently deprecated. Migrate existing Java code to use java.math.BigDecimal.

The new java.math.BigDecimal uses the same methods as both the previous java.math.BigDecimal and com.ibm.math.BigDecimal. Existing code using java.math.BigDecimal continues to work correctly. The two classes do not serialize.

To migrate existing Java code to use the java.math.BigDecimal class, change the import statement at the top of your `.java` file from: `import com.ibm.math.*;` to `import java.math.*;`.

Support for XToolkit

The IBM SDK for Linux, v6 includes XToolkit by default. You need XToolkit when using the SWT_AWT bridge in Eclipse to build an application that uses both SWT and Swing.

Restriction: Motif is no longer supported and will be removed in a later release.

Related links:

- An example of integrating Swing into Eclipse RCPs: http://eclipsezone.com/eclipse/forums/t45697.html
- Reference Information in the Eclipse information center: http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/awt/SWT_AWT.html
- Set up information is available on the Sun Microsystems Inc. Web site: http://java.sun.com/javase/6/docs/technotes/guides/awt/1.5/xawt.html

Support for the Java Attach API

The Java Attach API allows your application to connect to another virtual machine (the “target”). Your application can then load an agent application into the target virtual machine, for example to perform tasks such as monitoring status.

Code for agent applications, such as JMX agents or JVMTI agents, is normally loaded during virtual machine startup by specifying special startup parameters. Requiring startup parameters might not be convenient for using agents on

applications that are already running, such as WebSphere Application Servers. Using the Java Attach API, lets you load an agent at any time by specifying the process ID of the target virtual machine. The Attach API capability is sometimes called the “late attach” capability.

The Attach API is enabled by default for Java 6 SR 6 and later.

Security considerations

Security for the Java Attach API is handled by UNIX user and group file permissions.

The Java Attach API creates files and directories in a common directory. The common directory, subdirectories, and files in it, have UNIX file permissions. It is recommended that you change the ownership of the common directory to ROOT or another privileged user ID, to prevent 'spoofing' attacks.

The key security features of the Java Attach API are:

- A process using the Java Attach API must be owned by the same UNIX userid as the target process. This constraint ensures that only the target process owner can attach other applications to the target process.
- For Java 6 after SR 6, access to the files or directories owned by a process is controlled by user permissions only; group access is disabled.
- The common directory uses the sticky bit to prevent a user from deleting or replacing a subdirectory belonging to another user. To preserve the security of this mechanism, set the ownership of the common directory to ROOT.
- The subdirectory for a process is accessible only by members of the same UNIX group as the owner of a process. For Java 6 after SR 6, access is restricted to the owner only.
- Information about the target process can be written only by the owner, and read only by the owner or a member of the same group as the owner. For Java 6 after SR 6, access is restricted to the owner only.

You must secure access to the Java Attach API capability to ensure that only authorized users or processes can connect to another virtual machine. If you do not intend to use the Java Attach API capability, disable this feature using the Java system property. Set the **com.ibm.tools.attach.enable** system property to the value **no**; for example:

```
-Dcom.ibm.tools.attach.enable=no
```

Using the Java Attach API

By default, the target virtual machine is identified by its process ID. To use a different target, change the system property **com.ibm.tools.attach.id**; for example:

```
-Dcom.ibm.tools.attach.id=<process_ID>
```

The target process also has a human-readable “display name”. By default, the display name is the command line used to launch Java. To change the default display name, use the **com.ibm.tools.attach.displayName** system property. The ID and display name cannot be changed after the application has started.

The Attach API creates working files in a common directory called `.com_ibm_tools_attach`, which is created in the system temporary directory. The system property **java.io.tmpdir** holds the value of the system temporary directory.

On non-Windows systems, the system temporary directory is typically `/tmp`. To modify the working directory, use the Java system property **com.ibm.tools.attach.directory**; for example:

```
-Dcom.ibm.tools.attach.directory=/working
```

If your Java application ends abnormally, for example, following a crash or a SIGKILL signal, the process subdirectory is not deleted. The Java VM detects and removes obsolete subdirectories where possible. The subdirectory can also be deleted by the owning userid.

On heavily loaded system, applications might experience timeouts when attempting to connect to target applications. The default timeout is 120 seconds. Use the **com.ibm.tools.attach.timeout** system property to specify a different timeout value in seconds. For example, to timeout after 60 seconds:

```
-Dcom.ibm.tools.attach.timeout=60
```

A timeout value of zero indicates an indefinite wait.

For JMX applications, you might need to disable authentication by editing the `<JAVA_HOME>/jre/lib/management/management.properties` file. Set the following properties to disable authentication in JMX:

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

An unsuccessful attempt to invoke the Attach API results in one of the following exceptions:

- `com.ibm.tools.attach.AgentLoadException`
- `com.ibm.tools.attach.AgentInitializationException`
- `com.ibm.tools.attach.AgentNotSupportedException`
- `java.io.IOException`

A useful reference for information about the Attach API can be found at http://java.sun.com/javase/6/docs/technotes/guides/attach/index.html. The IBM implementation of the Attach API corresponds approximately to the Sun Microsystems, Inc. implementation. However, if your application originally used com.sun.tools.attach.* methods or classes, you must modify and recompile the application to use the com.ibm.tools.attach.* implementation.

Plug-in, Applet Viewer and Web Start

The Java plug-in is used to run Java applications in the browser. The **appletviewer** is used to test applications designed to be run in a browser. Java Web Start is used to deploy desktop Java applications over a network, and provides a mechanism for keeping them up-to-date.

Using the Java plug-in

The Java plug-in is a Web browser plug-in. You use the Java plug-in to run applets in the browser.

You must allow applets to finish loading to prevent your browser from stopping. For example, if you use the **Back** button and then the **Forward** button while an applet is loading, the HTML pages might be unable to load.

The Java plug-in is documented by Sun at: http://java.sun.com/javase/6/docs/technotes/guides/plugin/developer_guide/.

Supported browsers

The Java plug-in supports SeaMonkey, and Mozilla Firefox.

Table 25. Browsers supported by the Java plug-in on Linux IA32

Browser	Supported versions
Firefox	2.0, 3.0, 3.5

Table 26. Browsers supported by the Java plug-in on Linux PPC32

Browser	Supported Versions
SeaMonkey	1.0.8 **Note:** SeaMonkey is not supported on some Linux PPC32 operating systems.

Later minor releases of these browsers are also supported.

Installing the Java plug-in

To install the Java plug-in, symbolically link it to the plug-in directory for your browser.

The Java plug-in is based on Mozilla's Open JVM Integration initiative, which is used with most Mozilla products and derivatives, including Firefox.

You must symbolically link the plug-in, rather than copy it, so that the browser and plug-in can locate the JVM.

Firefox:

These steps make the Java plug-in available to Mozilla Firefox users.

Procedure

1. Log in as root.
2. Change to your Firefox plug-ins directory (which differs, depending on the Linux distributions).
   ```
   cd /usr/local/mozilla-firefox/plugins/
   ```
3. Create a symbolic link to the plug-in. To link with an old style plug-in:
   ```
   ln -s /opt/ibm/ibm-wrt-i386-60/jre/plugin/<arch>/ns7/libjavaplugin_oji.so .
   ```
   To link with a Next-Generation plug-in, which is available only with IA32:
   ```
   ln -s /opt/ibm/ibm-wrt-i386-60/jre/lib/<arch>/libnpjp2.so .
   ```
   Where i386 is the architecture of your system.

What to do next

You must symbolically link the plug-in, rather than copy it, so that it can locate the JVM.

Common Document Object Model (DOM) support

Because of limitations in particular browsers, you might not be able to implement all the functions of the org.w3c.dom.html package.

One of the following errors is thrown:

- sun.plugin.dom.exception.InvalidStateException
- sun.plugin.dom.exception.NotSupportedException

Using DBCS parameters

The Java plug-in supports double-byte characters (for example, Chinese Traditional BIG-5, Korean, and Japanese) as parameters for the tags <APPLET>, <OBJECT>, and <EMBED>. You must select the correct character encoding for your HTML document so that the Java plug-in can parse the parameter.

About this task

Specify character encoding for your HTML document by using the `<META>` tag in the `<HEAD>` section like this:

```
<meta http-equiv="Content-Type" content="text/html; charset=big5">
```

This example tells the browser to use the Chinese BIG-5 character encoding to parse the HTML file.

Working with applets

With the Applet Viewer, you can run one or more applets that are called by reference in a Web page (HTML file) by using the <APPLET> tag. The Applet Viewer finds the <APPLET> tags in the HTML file and runs the applets, in separate windows, as specified by the tags.

Because the Applet Viewer is for viewing applets, it cannot display a whole Web page that contains many HTML tags. It parses only the <APPLET> tags and no other HTML on the Web page.

Running and debugging applets with the Applet Viewer

Use the following commands to run and debug an applet with the Applet Viewer.

About this task

Running applets with the Applet Viewer:

From a shell prompt, enter:

```
appletviewer <name>
```

where <*name*> is one of the following:

- The file name of an HTML file that calls an applet.
- The URL of a Web page that calls an applet.

For example, to start the Applet Viewer on an HTML file that calls an applet, type at a shell prompt:

```
appletviewer $HOME/<filename>.html
```

Where *filename* is the name of the HTML file.

To start the Applet Viewer on a Web page, type at a shell prompt:

```
appletviewer http://java.sun.com/applets/jdk/1.4/demo/applets/NervousText/example1.html
```

The Applet Viewer does not recognize the **charset** option of the <

```
META
```

> tag. If the file that the Applet Viewer loads is not encoded as the system default, an I/O exception might occur. To avoid the exception, use the **-encoding** option when you run appletviewer. For example:

```
appletviewer -encoding JISAutoDetect sample.html
```

Debugging applets with the Applet Viewer:

For example:

```
cd demo/applets/TicTacToe
../../../bin/appletviewer -debug example1.html
```

You can find documentation about how to debug applets using the Applet Viewer at the Sun Web site: http://java.sun.com/javase/6/docs/technotes/guides/plugin/developer_guide/debugger.html.

Using Web Start

Java Web Start is used for Java application deployment.

With Web Start, you can start and manage applications directly from the Web. Applications are cached to minimize installation times. Applications are automatically upgraded when new versions become available.

Web Start supports these command-line arguments documented at http://java.sun.com/javase/6/docs/technotes/guides/javaws/developersguide/syntax.html#resources:

- -verbose
- -version
- -showversion
- -help
- -X
- -ea
- -enableassertions
- -da
- -disableassertions
- -esa
- -enablesystemassertions
- -dsa
- -disablesystemassertions
- -Xint
- -Xnoclassgc
- -Xdebug
- -Xfuture
- -Xrs
- -Xms
- -Xmx
- -Xss

Web Start also supports **-Xgcpolicy** to set the garbage collection policy.

For more information about Web Start, see:

- http://java.sun.com/products/javawebstart and
- http://java.sun.com/javase/6/docs/technotes/guides/javaws/index.html.

For more information about deploying applications, see:
- http://java.sun.com/javase/6/docs/technotes/guides/deployment/index.html.

Running Web Start
Web Start can be run from a Web page or the command line. Web Start applications are stored in the Java Application Cache.

About this task

You can start Web Start in a number of different ways.

Procedure
- Select a link on a Web page that refers to a `.jnlp` file. If your browser does not have the correct association to run Web Start applications, select the `/opt/ibm/ibm-wrt-i386-60/jre/bin/javaws` command from the **Open/Save** window to start the Web Start application.
- At a shell prompt, type:

  ```
  javaws <URL>
  ```

  Where <*URL*> is the location of a `.jnlp` file.
- If you have used Java Web Start to open the application in the past, use the Java Application Cache Viewer. At a shell prompt, type:

  ```
  /opt/ibm/ibm-wrt-i386-60/jre/bin/javaws -viewer
  ```

  All Java Web Start applications are stored in the Java Application Cache. An application is downloaded only if the latest version is not in the cache.

(Linux IA 32-bit only) WebStart Secure Static Versioning
Static versioning allows Web Start applications to request a specific JVM version on which those applications will run. Because static versioning also allows applications to exploit old security vulnerabilities on systems that have been upgraded to a new JVM, Secure Static Versioning (SSV) is now used by default.

With SSV, the user is warned before running any unsigned Web Start application that requests a specific JVM, if the requested JVM is installed. Signed applications and applications that request the latest version of the JVM run as usual.

You can disable SSV by setting the **deployment.javaws.ssv.enabled** property in the `deployment.properties` file to false.

Distributing Java applications
Java applications typically consist of class, resource, and data files.

When you distribute a Java application, your software package probably consists of the following parts:
- Your own class, resource, and data files
- An installation procedure or program

To run your application, a user needs the Runtime Environment for Linux. The SDK for Linux software contains a Runtime Environment. However, you cannot assume that your users have the SDK for Linux software installed.

Your SDK for Linux software license does **not** allow you to redistribute any of the SDK's files with your application. You must ensure that a licensed version of the SDK for Linux is installed on the target workstation.

Class data sharing between JVMs for non Real-Time

Class sharing is supported in non-real-time mode, but operates differently than in real-time mode.

The Java Virtual Machine (JVM) allows you to share class data between JVMs by storing it in a memory-mapped cache file on disk. Sharing reduces the overall virtual storage consumption when more than one JVM shares a cache. Sharing also reduces the startup time for a JVM after the cache has been created. The shared class cache is independent of any running JVM and persists until it is destroyed.

A shared cache can contain:

- Bootstrap classes
- Application classes
- Metadata that describes the classes
- Ahead-of-time (AOT) compiled code

Overview of class data sharing

Class data sharing provides a transparent method of reducing memory footprint and improving JVM start-up time. Java 6 provides new and improved features in cache management, isolation, and performance.

Enabling class data sharing

Enable class data sharing by using the **-Xshareclasses** option when starting a JVM. The JVM connects to an existing cache or creates a new cache if one does not exist.

All bootstrap and application classes loaded by the JVM are shared by default. Custom classloaders share classes automatically if they extend the application classloader; otherwise, they must use the Java Helper API provided with the JVM to access the cache. See “Adapting custom classloaders to share classes” on page 407.

The JVM can also store ahead-of-time (AOT) compiled code in the cache for certain methods to improve the startup time of subsequent JVMs. The AOT compiled code is not shared between JVMs, but is cached to reduce compilation time when the JVM starts up. The amount of AOT code stored in the cache is determined heuristically. You cannot control which methods get stored in the cache, but you can set upper and lower limits on the amount of cache space used for AOT code, or you can choose to disable AOT caching completely. See “Class data sharing command-line options” on page 400 for more information.

Cache access

A JVM can access a cache with either read-write or read-only access. Any JVM connected to a cache with read-write access can update the cache. Any number of JVMs can concurrently read from the cache, even while another JVM is writing to it.

You must take care if runtime bytecode modification is being used. See “Runtime bytecode modification” on page 406 for more information.

Dynamic updating of the cache

Because the shared class cache persists beyond the lifetime of any JVM, the cache is updated dynamically to reflect any modifications that might have been made to JARs or classes on the file system. The dynamic updating makes the cache transparent to the application using it.

Cache security

Access to the shared class cache is limited by operating system permissions and Java security permissions. The shared class cache is created with user access by default unless the **groupAccess** command-line suboption is used. Only a classloader that has registered to share class data can update the shared class cache.

The cache memory is protected against accidental or deliberate corruption using memory page protection. This protection is not an absolute guarantee against corruption because the JVM must unprotect pages to write to them. The only way to guarantee that a cache cannot be modified is to open it read-only.

If a Java SecurityManager is installed, classloaders, excluding the default bootstrap, application, and extension classloaders, must be granted permission to share classes by adding SharedClassPermission lines to the `java.policy` file. See “Using SharedClassPermission” on page 407. The RuntimePermission createClassLoader restricts the creation of new classloaders and therefore also restricts access to the cache.

Cache lifespan

Multiple caches can exist on a system and you specify them by name as a suboption to the **-Xshareclasses** command. A JVM can connect to only one cache at any one time.

You can override the default cache size on startup using **-Xscmx<n><size>**. This size is then fixed for the lifetime of the cache. Caches exist until they are explicitly destroyed using a suboption to the **-Xshareclasses** command or the cache file is deleted manually.

Cache utilities

All cache utilities are suboptions to the **-Xshareclasses** command. See “Class data sharing command-line options” or use **-Xshareclasses:help** to see a list of available suboptions.

Class data sharing command-line options

Class data sharing and the cache management utilities are controlled using command-line options to the Java launcher.

For options that take a <*size*> parameter, suffix the number with "k" or "K" to indicate kilobytes, "m" or "M" to indicate megabytes, or "g" or "G" to indicate gigabytes.

-Xscmaxaot<*size*>
Sets the maximum number of bytes in the cache that can be used for AOT data. Use this option to ensure a certain amount of cache space is available for non-AOT data. By default, the maximum limit for AOT data is the amount of free space in the cache. The value of this option should not be smaller than the value of **-Xscminaot** and must not be larger than the value of **-Xscmx**.

-Xscminaot<*size*>
Sets the minimum number of bytes in the cache to reserve for AOT data. By default, no space is reserved for AOT data, although AOT data is written to the cache until the cache is full or the **-Xscmaxaot** limit is reached. The value of this option must not exceed the value of **-Xscmx** or **-Xscmaxaot**. The value of **-Xscminaot** must always be considerably less than the total cache size because AOT data can be created only for cached classes. If the value of **-Xscminaot** is equal to the value of **-Xscmx**, no class data or AOT data is stored because AOT data must be associated with a class in the cache.

-Xscmx<*size*>
Specifies cache size. This option applies only if a cache is being created and no cache of the same name exists. The default cache size is platform-dependent. You can find out the size value being used by adding **-verbose:sizes** as a command-line argument. The minimum cache size is 4 KB. The maximum cache size is also platform-dependent. (See "Cache size limits" on page 405.)

-Xshareclasses:<*suboption*>**[,**<*suboption*>**...]**
Enables class data sharing. Can take a number of suboptions, some of which are cache utilities. Cache utilities perform the required operation on the specified cache, without starting the VM. You can combine multiple suboptions, separated by commas, but the cache utilities are mutually exclusive. When running cache utilities, the message `Could not create the Java virtual machine` is expected. Cache utilities do not create the virtual machine.

Some cache utilities can work with caches from previous Java versions or caches created by JVMs with different bit-widths. These caches are referred to as "incompatible" caches.

You can use the following suboptions with the **-Xshareclasses** option:

help
Lists all the command-line suboptions.

name=<*name*>
Connects to a cache of a given name, creating the cache if it does not already exist. Also used to indicate the cache that is to be modified by cache utilities; for example, **destroy**. Use the **listAllCaches** utility to show which named caches are currently available. If you do not specify a name, the default name "sharedcc_%u" is used. %u in the cache name inserts the current user name. You can specify "%g" in the cache name to insert the current group name.

cacheDir=<*directory*>
Sets the directory in which cache data is read and written. By default, <*directory*> is `/tmp/javasharedresources`. The user must have sufficient permissions in <*directory*>. The JVM writes persistent cache files directly into the directory specified. Persistent cache files can be safely moved and deleted from the file system. Non-persistent caches are stored in shared memory and have control files that describe the location of the memory. Control files are stored in a `javasharedresources` subdirectory of the **cacheDir** specified. Do not move or delete control files in this directory.

The **listAllCaches** utility, the **destroyAll** utility, and the **expire** suboption work only in the scope of a given **cacheDir**.

readonly
Opens an existing cache with read-only permissions. The JVM does not create a new cache with this suboption. Opening a cache read-only prevents the JVM from making any updates to the cache. It also allows the JVM to connect to caches created by other users or groups without requiring write access. By default, this suboption is not specified.

persistent (default)
Uses a persistent cache. The cache is created on disk, which persists beyond operating system restarts. Non-persistent and persistent caches can have the same name.

nonpersistent
Uses a non-persistent cache. The cache is created in shared memory, which is lost when the operating system shuts down. Non-persistent and persistent caches can have the same name. You must always use the **nonpersistent** suboption when running utilities such as **destroy** on a non-persistent cache.

groupAccess
Sets operating system permissions on a new cache to allow group access to the cache. The default is user access only.

verbose
Enables verbose output, which provides overall status on the shared class cache and more detailed error messages.

verboseAOT
Enables verbose output when compiled AOT code is being found or stored in the cache. AOT code is generated heuristically. You might not see any AOT code generated at all for a small application. You can disable AOT caching using the **noaot** suboption.

verboseIO
Gives detailed output on the cache I/O activity, listing information on classes being stored and found. Each classloader is given a unique ID (the bootstrap loader is always 0) and the output shows the classloader hierarchy at work, where classloaders must ask their parents for a class before they can load it themselves. It is usual to see many failed requests; this behavior is expected for the classloader hierarchy.

verboseHelper
Enables verbose output for the Java Helper API. This output shows you how the Helper API is used by your ClassLoader.

silent
Turns off all shared classes messages, including error messages. Unrecoverable error messages, which prevent the JVM from initializing, are displayed.

nonfatal
Allows the JVM to start even if class data sharing fails. Normal behavior for the JVM is to refuse to start if class data sharing fails. If you select **nonfatal** and the shared classes cache fails to initialize, the JVM attempts to connect to the cache in read-only mode. If this attempt fails, the JVM starts without class data sharing.

none
Can be added to the end of a command line to disable class data sharing. This suboption overrides class sharing arguments found earlier on the command line.

modified=<*modified context*>
Used when a JVMTI agent is installed that might modify bytecode at runtime. If you do not specify this suboption and a bytecode modification agent is installed, classes are safely shared with an extra performance cost. The <*modified context*> is a descriptor chosen by the user; for example, "myModification1". This option partitions the cache, so that only JVMs using context myModification1 can share the same classes. For instance, if you run HelloWorld with a modification context and then run it again with a different modification context, all classes are stored twice in the cache. See "Runtime bytecode modification" on page 406 for more information.

reset
Causes a cache to be destroyed and then recreated when the JVM starts up. Can be added to the end of a command line as **-Xshareclasses:reset**.

destroy (Utility option)
Destroys a cache specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. A cache can be destroyed only if all JVMs using it have shut down, and the user has sufficient permissions.

destroyAll (Utility option)
Tries to destroy all caches available using the specified **cacheDir** and **nonpersistent** suboptions. A cache can be destroyed only if all JVMs using it have shut down, and the user has sufficient permissions.

expire=<*time in minutes*>
Destroys all caches that have been unused for the time specified before loading shared classes. This option is not a utility option because it does not cause the JVM to exit.

listAllCaches (Utility option)
Lists all the compatible and incompatible caches that exist in the specified cache directory. If you do not specify **cacheDir**, the default directory is used. Summary information, such as Java version and current usage is displayed for each cache.

printStats (Utility option)
Displays summary information for the cache specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. The most useful information displayed is how full the cache is and how many classes it contains. Stale classes are classes that have been updated on the file system and which the cache has therefore marked "stale". Stale classes are not purged from the cache and can be reused. See the Diagnostics Guide for more information.

printAllStats (Utility option)
Displays detailed information for the cache specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. Every class is listed in chronological order, with a reference to the location from which it was loaded. AOT code for class methods is also listed.

See the Diagnostics Guide for more information.

mprotect=[all | default | none]
By default, the memory pages containing the cache are protected at all times, unless a specific page is being updated. This protection helps prevent accidental or deliberate corruption to the cache. The cache header

is not protected by default because this protection has a small performance cost. Specifying `all` ensures that all the cache pages are protected, including the header. Specifying `none` disables the page protection.

noBootclasspath
Prevents storage of classes loaded by the bootstrap classloader in the shared classes cache. Can be used with the SharedClassURLFilter API to control exactly which classes get cached. See the Diagnostics Guide for more information about shared class filtering.

cacheRetransformed
Enables caching of classes that have been transformed using the JVMTI RetransformClasses function.

noaot
Disables caching of AOT code. AOT code already in the shared data cache can be loaded.

Creating, populating, monitoring, and deleting a cache

An overview of the life-cycle of a shared class data cache including examples of the cache management utilities.

To enable class data sharing, add **-Xshareclasses[:name=<name>]** to your application command line.

The JVM either connects to an existing cache of the given name or creates a new cache of that name. If a new cache is created, it is populated with all bootstrap and application classes being loaded until the cache becomes full. If two or more JVMs are started concurrently, they populate the cache concurrently.

To check that the cache has been created, run `java -Xshareclasses:listAllCaches`. To see how many classes and how much class data is being shared, run `java -Xshareclasses:[name=<name>],printStats`. You can run these utilities after the application JVM has terminated or in another command window.

For more feedback on cache usage while the JVM is running, use the **verbose** suboption. For example, `java -Xshareclasses:[name=<name>],verbose`.

To see classes being loaded from the cache or stored in the cache, add `-Xshareclasses:[name=<name>],verboseIO` to your application command line.

To delete the cache, run `java -Xshareclasses:[name=<name>],destroy`. You usually delete caches only if they contain many stale classes or if the cache is full and you want to create a bigger cache.

You should tune the cache size for your specific application, because the default is unlikely to be the optimum size. To determine the optimum cache size, specify a large cache, using **-Xscmx**, run the application, and then use **printStats** to determine how much class data has been stored. Add a small amount to the value shown in **printStats** for contingency. Because classes can be loaded at any time during the lifetime of the JVM, it is best to do this analysis after the application has terminated. However, a full cache does not have a negative affect on the performance or capability of any JVMs connected to it, so it is acceptable to decide on a cache size that is smaller than required.

If a cache becomes full, a message is displayed on the command line of any JVMs using the **verbose** suboption. All JVMs sharing the full cache then loads any

further classes into their own process memory. Classes in a full cache can still be shared, but a full cache is read-only and cannot be updated with new classes.

Performance and memory consumption

Class data sharing is particularly useful on systems that use more than one JVM running similar code; the system benefits from reduced virtual storage consumption. It is also useful on systems that frequently start up and shut down JVMs, which benefit from the improvement in startup time.

The processor and memory usage required to create and populate a new cache is minimal. The JVM startup cost in time for a single JVM is typically between 0 and 5% slower compared with a system not using class data sharing, depending on how many classes are loaded. JVM startup time improvement with a populated cache is typically between 10% and 40% faster compared with a system not using class data sharing, depending on the operating system and the number of classes loaded. Multiple JVMs running concurrently show greater overall startup time benefits.

Duplicate classes are consolidated in the shared class cache. For example, class A loaded from `myClasses.jar` and class A loaded from `myOtherClasses.jar` (with identical content) is stored only once in the cache. The **printAllStats** utility shows multiple entries for duplicated classes, with each entry pointing to the same class.

When you run your application with class data sharing, you can use the operating system tools to see the reduction in virtual storage consumption.

Considerations and limitations of using class data sharing

Consider these factors when deploying class data sharing in a product and using class data sharing in a development environment.

Cache size limits

The maximum theoretical cache size is 2 GB. The size of cache you can specify is limited by the amount of physical memory and paging space available to the system.

The shared class cache consists of memory mapped files that are created on disk and remain when the operating system is restarted. If you change the default behavior using the **-Xshareclasses:nonpersistent** option, so that the cache is not retained on restart, the cache for sharing classes is allocated using the System V IPC shared memory mechanism. In this case, cache size is limited by **SHMMAX** settings, which limits the amount of shared memory that can be allocated. You can find these settings by looking at the`/proc/sys/kernel/shmmax` file. **SHMMAX** is typically set to 30 MB.

Because the virtual address space of a process is shared between the shared classes cache and the Java heap, if you increase the maximum size of the Java heap you might reduce the size of the shared classes cache you can create.

JVMTI RetransformClasses() is unsupported

You cannot run RetransformClasses() on classes loaded from the shared class cache.

The JVM might throw the exception UnmodifiableClassException if you attempt to run RetransformClasses(). It does not work because class file bytes are not available for classes loaded from the shared class cache. If you must use

RetransformClasses(), ensure that the classes to be transformed are not loaded from the shared class cache, or disable the shared class cache feature.

Runtime bytecode modification

Any JVM using a JVM Tool Interface (JVMTI) agent that can modify bytecode data must use the **modified=**<*modified_context*> suboption if it wants to share the modified classes with another JVM.

The modified context is a user-specified descriptor that describes the type of modification being performed. The modified context partitions the cache so that all JVMs running under the same context share a partition.

This partitioning allows JVMs that are not using modified bytecode to safely share a cache with those that are using modified bytecode. All JVMs using a given modified context must modify bytecode in a predictable, repeatable manner for each class, so that the modified classes stored in the cache have the expected modifications when they are loaded by another JVM. Any modification must be predictable because classes loaded from the shared class cache cannot be modified again by the agent.

If a JVMTI agent is used without a modification context, classes are still safely shared by the JVM, but with a small affect on performance. Using a modification context with a JVMTI agent avoids the need for extra checks and therefore has no affect on performance. A custom ClassLoader that extends java.net.URLClassLoader and modifies bytecode at load time without using JVMTI automatically stores that modified bytecode in the cache, but the cache does not treat the bytecode as modified. Any other VM sharing that cache loads the modified classes. You can use the **modified=**<*modification_context*> suboption in the same way as with JVMTI agents to partition modified bytecode in the cache. If a custom ClassLoader needs to make unpredictable load-time modifications to classes, that ClassLoader must not attempt to use class data sharing.

See the Diagnostics Guide for more detail on this topic.

Operating system limitations

You cannot share classes between 32-bit and 64-bit JVMs. Temporary disk space must be available to hold cache information. The operating system enforces cache permissions.

For operating systems that can run both 32-bit and 64-bit applications, class data sharing is not permitted between 32-bit and 64-bit JVMs. The **listAllCaches** suboption lists 32-bit or 64-bit caches, depending on the address mode of the JVM being used.

The shared class cache requires disk space to store identification information about the caches that exist on the system. This information is stored in `/tmp/javasharedresources`. If the identification information directory is deleted, the JVM cannot identify the shared classes on the system and must re-create the cache. Use the ipcs command to view the memory segments used by a JVM or application.

Users running a JVM must be in the same group to use a shared class cache. The operating system enforces the permissions for accessing a shared class cache. If you do not specify a cache name, the user name is appended to the default name so that multiple users on the same system create their own caches by default.

Using SharedClassPermission

If a SecurityManager is being used with class data sharing and the running application uses its own class loaders, you must grant these class loaders shared class permissions before they can share classes.

You add shared class permissions to the `java.policy` file using the ClassLoader class name (wildcards are permitted) and either "read", "write", or "read,write" to determine the access granted. For example:

```
permission com.ibm.oti.shared.SharedClassPermission
        "com.abc.customclassloaders.*", "read,write";
```

If a ClassLoader does not have the correct permissions, it is prevented from sharing classes. You cannot change the permissions of the default bootstrap, application, or extension class loaders.

Adapting custom classloaders to share classes

Any classloader that extends java.net.URLClassLoader can share classes without modification. You must adopt classloaders that do not extend java.net.URLClassLoader to share class data.

You must grant all custom classloaders shared class permissions if a SecurityManager is being used; see "Using SharedClassPermission." IBM provides several Java interfaces for various types of custom classloaders, which allow the classloaders to find and store classes in the shared class cache. These classes are in the com.ibm.oti.shared package.

The Javadoc document for this package is provided with the SDK in the `docs/content/apidoc` directory.

See the Diagnostics Guide for more information about how to use these interfaces.

Java Communications API (JavaComm)

The Java Communications (API) package (JavaComm) is an optional package provided for use with the Runtime Environment for Linux on the IA32, PPC32/PPC64, and AMD64/EM64T platforms. You install JavaComm independently of the SDK or Runtime Environment.

The JavaComm API gives Java applications a platform-independent way of performing serial and parallel port communications for technologies such as voice mail, fax, and smartcards.

The Java Communications API supports Electronic Industries Association (EIA)-232 (RS232) serial ports and Institute of Electrical and Electronics Engineers (IEEE) 1284 parallel ports and is supported on systems with the IBM Version 6 Runtime Environment.

Using the Java Communications API, you can:

- List ports on a system
- Open and claim ownership of ports
- Resolve port ownership contention among applications that use Java Communications API
- Perform asynchronous and synchronous I/O port-monitoring using event notification

- Receive bean-style events describing state changes on the port

Installing Java Communications API from a compressed file

Make sure that the SDK or Runtime Environment is installed before you install the Java Communications API.

About this task

If you used the RPM package to install Java originally, install the Java Communications API from the RPM file. To install the Java Communications API from an RPM package, see "Installing the Java Communications API from an RPM file."

To install the Java Communications API from a compressed file:

Procedure

1. Download the Java Communications API compressed file from http://www.ibm.com/developerworks/java/jdk/linux/download.html.
2. Put the Java Communications API compressed file, `ibm-java-javacomm-60–3.0–0.0-linux-i386.tar.gz`, in the directory where the SDK or Runtime Environment is installed. If you installed to the default directory, this is `/opt/ibm/ibm-wrt-i386-60/`.
3. From a shell prompt, in the directory containing the compressed file, extract the contents:

   ```
   tar -xvzf ibm-java-javacomm-60–3.0–0.0-linux-i386.tar.gz
   ```

4. Copy the Java Communications API files into the correct directories in your SDK.
   a. Copy `lib/libLinuxSerialParallel.so` to your `jre/lib/<arch>/` directory. Where *<arch>* is the architecture of your platform.
   b. Copy `jar/comm.jar` to your `jre/lib/ext/` directory.
   c. Copy `lib/javax.comm.properties` to your `jre/lib/` directory.

   By default, the SDK is installed in the /opt/ibm/ibm-wrt-i386-60/ directory.

Installing the Java Communications API from an RPM file

Make sure that a copy of the SDK or Runtime Environment is installed before you install the Java Communications API.

About this task

If you used the RPM package to install Java originally, install the Java Communications API from the RPM file.

Procedure

1. Download the Java Communications API RPM package from http://www.ibm.com/developerworks/java/jdk/linux/download.html.
2. Open a shell prompt, making sure you are root.
3. Use the `rpm -ivh` command to install the Java Communications API RPM file. For example:

   ```
   rpm -ivh ibm-java-i386-javacomm-5.0-0.0.i386.rpm
   ```

   The Java Communications API is installed in the /opt/ibm/ibm-wrt-i386-60/ directory structure.

4. Copy the javacomm files into the correct directories in your SDK.
   a. Copy `lib/libLinuxSerialParallel.so` to your `jre/lib/<arch>/` directory. Where *<arch>* is the architecture of your platform.
   b. Copy `jar/comm.jar` to your `jre/lib/ext/` directory.
   c. Copy `lib/javax.comm.properties` to your `jre/lib/` directory.

   By default, the SDK is installed in the `/opt/ibm/ibm-wrt-i386-60/` directory.

Location of the Java Communications API files

By default, the Java Communications API files are installed in the `/opt/ibm/ibm-wrt-i386-60/` directory.

The files and their structure are:

- `jar/comm.jar`
- `jar/commtest.jar`
- `jar/tools/BlackBox.jar`
- `jar/tools/ParallelBlackBox.jar`
- `lib/javax.comm.properties`
- `lib/libLinuxSerialParallel.so`

Configuring the Java Communications API

To use the Java Communications API, you must change the access mode of serial and parallel ports, and set the **PATH** if you did not set it when you installed Java.

About this task

See "Setting the path" on page 364.

Changing the access mode of serial and parallel ports

After you install Java Communications API, you must change the access mode of serial and parallel ports so that users can access these devices.

About this task

You must give a user read and write access to the required devices. Log on as root and use the following commands, as applicable:

```
chmod 660 /dev/ttyS0    (serial port COM1)
chmod 660 /dev/ttyS1    (serial port COM2)
chmod 660 /dev/ttyS2    (serial port COM3)
chmod 660 /dev/ttyS3    (serial port COM4)
chmod 660 /dev/parport0 (raw parallel ports)
chmod 660 /dev/parport1 (raw parallel ports)
```

Add specific users to the same group that the devices are in. On a SUSE system, for example, the devices are in the `uucp` group. Thus, users can be added to the `uucp` group to gain access to the devices.

Change the access mode of any other ports as needed.

Specifying devices in the `javax.comm.properties` file

Use the javax.comm.properties file to specify the devices and drivers that are available to the Java Communications API and whether they are parallel or serial. Do not change this file without a very clear understanding of its use.

About this task

The following properties must be defined:

```
driver=<driver_name>
serpath0=<serial_port_device>
parpath0=<parallel_port_device>
```

For example:

```
# Implementation specific driver
driver=com.sun.comm.LinuxDriver

# Paths to server-side serial port devices
serpath0 = /dev/ttyS0
serpath1 = /dev/ttyS1

# Paths to server-side parallel port devices
parpath0 = /dev/parport0
parpath1 = /dev/parport1
```

For parallel port access, use `/dev/parport<n>`. `/dev/lp<n>` is not supported in Javacomm version 3.0.

Enabling serial ports on IBM ThinkPads

Most ThinkPads have their serial ports disabled by default in the BIOS. Currently, there is no way to enable the ports with Linux (the tpctl package *does not* enable the ports if they are disabled in the BIOS).

About this task

To enable the ports in the BIOS, you must use the DOS version of the ThinkPad Configuration Utility that is available from the IBM ThinkPad Download site. To use the ThinkPad Configuration Utility, you need a bootable DOS diskette. The ThinkPad Configuration Utility might have been installed as part of the ThinkPad Utilities under Windows, depending on your installation options, and you can run it from a command prompt in Windows.

The ThinkPad Configuration application provided with Windows has options to enable or disable the serial and parallel ports but this *does not* also change the settings in the BIOS. So if you use this application with Windows, the ports are available; however, if you reboot your system with Linux, the ports *will not* be enabled.

Printing limitation with the Java Communications API

When printing with the Java Communications API, you might have to select “Form feed”, “Continue”, or a similar option on the printer.

Uninstalling Java Communications API

The process you use to uninstall the Java Communications API depends on whether you installed the installable Red Hat Package Manager (RPM) package or the compressed Tape Archive (TAR) package.

About this task

Uninstalling the Red Hat Package Manager (RPM) package

Uninstalling the Java Communications API using the RPM package.

About this task

Procedure

1. Use the rpm tool to uninstall the package. Alternatively, you can use a graphical tool such as kpackage or yast2.
2. If the directory where you installed the Java Communications API does not contain any other tools that you require, remove that directory from your **PATH** statement.
3. If you copied the Java communications API libraries into the SDK directory, delete the following files from the SDK directory.
   - `jre/lib/<arch>/libLinuxSerialParallel.so`
   - `jre/lib/ext/comm.jar`
   - `jre/lib/javax.comm.properties`

   Where *<arch>* is the architecture of your platform. By default, the SDK is installed in the /opt/ibm/ibm-wrt-i386-60/ directory.

Uninstalling the compressed Tape Archive (TAR) package

Uninstalling the Java Communications API, if you installed the compressed TAR package.

About this task

Delete the following files from the directory where you installed them:

- `jre/lib/<arch>/libLinuxSerialParallel.so`
- `jre/lib/ext/comm.jar`
- `jre/lib/javax.comm.properties`

Where *<arch>* is the architecture of your platform.

The Java Communications API documentation

You can find API documentation and samples for the Java Communications API at the Sun Web site.

http://java.sun.com/products/javacomm/.

Service and support for independent software vendors

Contact points for service:

If you are entitled to services for the Program code pursuant to the IBM Solutions Developer Program, contact the IBM Solutions Developer Program through your usual method of access or on the Web at: http://www.ibm.com/partnerworld/.

If you have purchased a service contract (that is, the IBM Personal Systems Support Line or equivalent service by country), the terms and conditions of that service contract determine what services, if any, you are entitled to receive with respect to the Program.

Accessibility

The user guides that are supplied with this SDK and the Runtime Environment have been tested using screen readers.

To change the font sizes in the user guides, use the function that is supplied with your browser, typically found under the **View** menu option.

For users who require keyboard navigation, a description of useful keystrokes for Swing applications is in *Swing Key Bindings* at http://www.ibm.com/developerworks/java/jdk/additional/.

Keyboard traversal of JComboBox components in Swing

If you traverse the drop-down list of a JComboBox component with the cursor keys, the button or editable field of the JComboBox does not change value until an item is selected. This is the correct behavior for this release and improves accessibility and usability by ensuring that the keyboard traversal behavior is consistent with mouse traversal behavior.

Web Start accessibility (Linux IA 32-bit, PPC32, and PPC64 only)

From Version 5.0, Java Web Start contains several accessibility and usability improvements, including better support for screen readers and improved keyboard navigation.

You can use the command line to start a Java application that is enabled for Web Start. To change preference options, you must edit a configuration file, `.java/.deployment/.deployment.properties` in the user's home directory. Take a backup before you edit this file. Not all of the preferences that can be set in the Java Application Cache Viewer are available in the configuration file.

Appendixes

Reference information.

Command-line options

You can specify the options on the command line while you are starting Java. They override any relevant environment variables. For example, using **-cp <dir1>** with the Java command completely overrides setting the environment variable **CLASSPATH=<dir2>**.

This chapter provides the following information:

Specifying command-line options

Although the command line is the traditional way to specify command-line options, you can pass options to the JVM in other ways.

Use only single or double quotation marks for command-line options when explicitly directed to do so for the option in question. Single and double quotation marks have different meanings on different platforms, operating systems, and

shells. Do not use **'-X<option>'** or **"-X<option>"**. Instead, you must use **-X<option>**. For example, do not use `'-Xmx500m'` and `"-Xmx500m"`. Write this option as `-Xmx500m`.

These precedence rules (in descending order) apply to specifying options:

1. Command line.

   For example, `java -X<option> MyClass`
2. A file containing a list of options, specified using the **–Xoptionsfile** option on the command line. For example, `java -Xoptionsfile=myoptionfile.txt MyClass`

   In the options file, specify each option on a new line; you can use the '\' character as a continuation character if you want a single option to span multiple lines. Use the '#' character to define comment lines. You cannot specify **-classpath** in an options file. Here is an example of an options file:

   ```
   #My options file
   -X<option1>
   -X<option2>=\
   <value1>,\
   <value2>
   -D<sysprop1>=<value1>
   ```
3. **IBM_JAVA_OPTIONS** environment variable. You can set command-line options using this environment variable. The options that you specify with this environment variable are added to the command line when a JVM starts in that environment.

   For example, `set IBM_JAVA_OPTIONS=-X<option1> -X<option2>=<value1>`

General command-line options

Use these options to print help on assert-related options, set the search path for application classes and resources, print a usage method, identify memory leaks inside the JVM, print the product version and continue, enable verbose output, and print the product version.

-cp, -classpath <directories and compressed or jar files separated by : (; on Windows)>
Sets the search path for application classes and resources. If **-classpath** and **-cp** are not used, and the **CLASSPATH** environment variable is not set, the user classpath is, by default, the current directory (`.`).

-help, -?
Prints a usage message.

-showversion
Prints product version and continues.

-verbose:<*option*>**[,**<*option*>**...]**
Enables verbose output. Separate multiple options using commas. These options are available:

class
Writes an entry to `stderr` for each class that is loaded.

dynload
Provides detailed information as each bootstrap class is loaded by the JVM:

- The class name and package
- For class files that were in a .jar file, the name and directory path of the .jar
- Details of the size of the class and the time taken to load the class

The data is written out to `stderr`. An example of the output on a Windows platform follows:

```
<Loaded java/lang/String from C:\sdk\jre\lib\vm.jar>
<Class size 17258; ROM size 21080; debug size 0>
<Read time 27368 usec; Load time 782 usec; Translate time 927 usec>
```

gc Provide verbose garbage collection information.

init
Writes information to `stderr` describing JVM initialization and termination.

jni
Writes information to `stderr` describing the JNI services called by the application and JVM.

sizes
Writes information to `stderr` describing the active memory usage settings.

stack
Writes information to `stderr` describing the Java and C stack usage for each thread.

-version
Prints product version.

System property command-line options

Use the system property command-line options to set up your system.

-D<*name*>=<*value*>
Sets a system property.

-Dcom.ibm.jsse2.renegotiate=[ALL | NONE | ABBREVIATED]
If your Java application uses JSSE for secure communication, you can disable TLS renegotiation by installing APAR IZ65239.

ALL Allow both abbreviated and unabbreviated (full) renegotiation handshakes.

NONE
Allow no renegotiation handshakes. This is the default setting.

ABBREVIATED
Allow only abbreviated renegotiation handshakes.

-Dcom.ibm.lang.management.verbose
Enables verbose information from java.lang.management operations to be written to the console during VM operation.

-Dcom.ibm.tools.attach.enable=yes
Enable the Attach API for this application. The Attach API allows your application to connect to a virtual machine. Your application can then load an agent application into the virtual machine. The agent can be used to perform tasks such as monitoring the virtual machine status.

-Dibm.jvm.bootclasspath
The value of this property is used as an additional search path, which is inserted between any value that is defined by **-Xbootclasspath/p:** and the bootclass path. The bootclass path is either the default or the one that you defined by using the **-Xbootclasspath:** option.

-Dibm.stream.nio=[true | false]
From v1.4.1 onwards, by default the IO converters are used. This option addresses the ordering of IO and NIO converters. When this option is set to true, the NIO converters are used instead of the IO converters.

-Djava.compiler=[NONE | j9jit24]
Disables the Java compiler by setting to NONE. Enable JIT compilation by setting to j9jit24 (Equivalent to **–Xjit**).

-Djava.net.connectiontimeout=[n]
'n' is the number of seconds to wait for the connection to be established with the server. If this option is not specified in the command line, the default value of 0 (infinity) is used. The value can be used as a timeout limit when an asynchronous java-net application is trying to establish a connection with its server. If this value is not set, a java-net application waits until the default connection timeout value is met. For instance, `java -Djava.net.connectiontimeout=2 TestConnect` causes the java-net client application to wait only 2 seconds to establish a connection with its server.

-Dsun.net.client.defaultConnectTimeout=<*value in milliseconds*>
Specifies the default value for the connect timeout for the protocol handlers used by the java.net.URLConnection class. The default value set by the protocol handlers is -1, which means that no timeout is set.

When a connection is made by an applet to a server and the server does not respond properly, the applet might seem to hang and might also cause the browser to hang. This apparent hang occurs because there is no network connection timeout. To avoid this problem, the Java Plug-in has added a default value to the network timeout of 2 minutes for all HTTP connections. You can override the default by setting this property.

-Dsun.net.client.defaultReadTimeout=<*value in milliseconds*>
Specifies the default value for the read timeout for the protocol handlers used by the java.net.URLConnection class when reading from an input stream when a connection is established to a resource. The default value set by the protocol handlers is -1, which means that no timeout is set.

-Dsun.nio.MaxDirectMemorySize=<*value in bytes*>
Limits the native memory size for nio Direct Byte Buffer objects to the value specified.

-Dsun.rmi.transport.tcp.connectionPool=[true | any non-null value]
Enables thread pooling for the RMI ConnectionHandlers in the TCP transport layer implementation.

-Dswing.useSystemFontSettings=[false]
From v1.4.1 onwards, by default, Swing programs running with the Windows Look and Feel render with the system font set by the user instead of a Java-defined font. As a result, fonts for v1.4.1 differ from those in earlier releases. This option addresses compatibility problems like these for programs that depend on the old behavior. By setting this option, v1.4.1 fonts and those of earlier releases are the same for Swing programs running with the Windows Look and Feel.

JVM command-line options

Use these options to configure your JVM. The options prefixed with **-X** are nonstandard.

For options that take a <*size*> parameter, suffix the number with "k" or "K" to indicate kilobytes, "m" or "M" to indicate megabytes, or "g" or "G" to indicate gigabytes.

For options that take a <*percentage*> parameter, use a number from 0 to 1. For example, 50% is 0.5.

Options that relate to the JIT are listed under "JIT and AOT command-line options" on page 426. Options that relate to the Garbage Collector are listed under "Garbage Collector command-line options" on page 428.

-X Displays help on nonstandard options.

-Xaggressive
(Linux PPC32 only) Enables performance optimizations that are expected to be the default in future releases.

-Xargencoding
You can put Unicode escape sequences in the argument list. This option is set to off by default.

-Xbootclasspath:<*directories and compressed or Java archive files separated by : (; on* **Windows***)*>
Sets the search path for bootstrap classes and resources. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xbootclasspath/a:<*directories and compressed or Java archive files separated by : (; on* **Windows***)*>
Appends the specified directories, compressed files, or jar files to the end of the bootstrap class path. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xbootclasspath/p:<*directories and compressed or Java archive files separated by : (; on* **Windows***)*>
Adds a prefix of the specified directories, compressed files, or Java archive files to the front of the bootstrap class path. Do not deploy applications that use the **-Xbootclasspath:** or the **-Xbootclasspath/p:** option to override a class in the standard API. This is because such a deployment contravenes the Java 2 Runtime Environment binary code license. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xcheck:classpath
Displays a warning message if an error is discovered in the class path; for example, a missing directory or JAR file.

-Xcheck:gc[:<*scan options*>][:<*verify options*>][:<*misc options*>]
Performs additional checks on garbage collection. By default, no checking is performed. See the output of **-Xcheck:gc:help** for more information.

-Xcheck:jni[**:help**][:<*option*>=<*value*>]
Performs additional checks for JNI functions. This option is equivalent to **-Xrunjnichk**. By default, no checking is performed.

-Xcheck:memory[:<*option*>]
Identifies memory leaks inside the JVM using strict checks that cause the JVM to exit on failure. If no option is specified, **all** is used by default. The available options are as follows:

all Enables checking of all allocated and freed blocks on every free and allocate call. This check of the heap is the most thorough. It typically causes the JVM to exit on nearly all memory-related problems soon after they are caused. This option has the greatest affect on performance.

callsite=<*number of allocations*>

Displays callsite information every <*number of allocations*>. Deallocations are not counted. Callsite information is presented in a table with separate information for each callsite. Statistics include:

- The number and size of allocation and free requests since the last report.
- The number of the allocation request responsible for the largest allocation from each site.

Callsites are presented as `sourcefile:linenumber` for C code and assembly function name for assembler code.

Callsites that do not provide callsite information are accumulated into an "unknown" entry.

failat=<*number of allocations*>
Causes memory allocation to fail (return NULL) after <*number of allocations*>. Setting <*number of allocations*> to 13 causes the 14th allocation to return NULL. Deallocations are not counted. Use this option to ensure that JVM code reliably handles allocation failures. This option is useful for checking allocation site behavior rather than setting a specific allocation limit.

ignoreUnknownBlocks
Ignores attempts to free memory that was not allocated using the **-Xcheck:memory** tool. Instead, the **-Xcheck:memory** statistics printed out at the end of a run indicates the number of “unknown” blocks that were freed.

mprotect=<*top*|*bottom*>
Locks pages of memory on supported platforms, causing the program to stop if padding before or after the allocated block is accessed for reads or writes. An extra page is locked on each side of the block returned to the user.

If you do not request an exact multiple of one page of memory, a region on one side of your memory is not locked. The top and bottom options control which side of the memory area is locked. top aligns your memory blocks to the top of the page, so buffer underruns result in an application failure. bottom aligns your memory blocks to the bottom of the page so buffer overruns result in an application failure.

Standard padding scans detect buffer underruns when using top and buffer overruns when using bottom.

nofree
Keeps a list of blocks already used instead of freeing memory. This list is checked, as well as currently allocated blocks, for memory corruption on every allocation and deallocation. Use this option to detect a dangling pointer (a pointer that is "dereferenced" after its target memory is freed). This option cannot be reliably used with long-running applications (such as WebSphere Application Server), because “freed” memory is never reused or released by the JVM.

noscan
Checks for blocks that are not freed. This option has little effect on performance, but memory corruption is not detected. This option is compatible only with **subAllocator**, **callsite**, and **callsitesmall**.

quick
Enables block padding only and is used to detect basic heap corruption. Every allocated block is padded with sentinel bytes, which are verified on every allocate and free. Block padding is faster than the default of checking every block, but is not as effective.

skipto=<*number of allocations*>
Causes the program to check only on allocations that occur after <*number of allocations*>. Deallocations are not counted. Use this option to speed up JVM startup when early allocations are not causing the memory problem. The JVM performs approximately 250+ allocations during startup.

subAllocator[=<*size in MB*>]
Allocates a dedicated and contiguous region of memory for all JVM allocations. This option helps to determine if user JNI code or the JVM is responsible for memory corruption. Corruption in the JVM **subAllocator** heap suggests that the JVM is causing the problem; corruption in the user-allocated memory suggests that user code is corrupting memory. Typically, user and JVM allocated memory are interleaved.

zero
Newly allocated blocks are set to 0 instead of being filled with the `0xE7E7xxxxxxxxE7E7` pattern. Setting these blocks to 0 helps you to determine whether a callsite is expecting zeroed memory, in which case the allocation request is followed by `memset(pointer, 0, size)`.

-Xclassgc
Enables dynamic unloading of classes by the JVM. This is the default behavior. To disable dynamic class unloading, use the **-Xnoclassgc** option.

-Xcompressedrefs
(64-bit only) Uses 32-bit values for references. See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html) section on “Compressed references” for more information. By default, references are 64-bit.

-Xdbg:<*options*>
Loads debugging libraries to support the remote debugging of applications. This option is equivalent to **-Xrunjdwp**. By default, the debugging libraries are not loaded, and the VM instance is not enabled for debug.

-Xdebug
This option is deprecated. Use **-Xdbg** for debugging.

-Xdisablejavadump
Turns off Javadump generation on errors and signals. By default, Javadump generation is enabled.

-Xdump
See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html) section on “Using dump agents” for more information.

-Xenableexplicitgc
Signals to the VM that calls to System.gc() trigger a garbage collection. This option is enabled by default.

-Xfuture
Turns on strict class-file format checks. Use this flag when you are developing new code because stricter checks will become the default in future releases. By default, strict format checks are disabled.

-Xiss<*size*>
Sets the initial stack size for Java threads. By default, the stack size is set to 2 KB. Use the **-verbose:sizes** option to output the value that the VM is using.

-Xjarversion
Produces output information about the version of each jar file in the class path,

the boot class path, and the extensions directory. Version information is taken from the Implementation-Version and Build-Level properties in the manifest of the jar.

-Xjni:<*suboptions*>
Sets JNI options. You can use the following suboption with the **-Xjni** option:

-Xjni:arrayCacheMax=[<*size in bytes*>**| unlimited]**
Sets the maximum size of the array cache. The default size is 8096 bytes.

-Xlinenumbers
Displays line numbers in stack traces for debugging. See also **-Xnolinenumbers**. By default, line numbers are on.

-Xlog

Enables message logging. To prevent message logging, use the **-Xlog:none** option. By default, logging is enabled.

-Xlp<*size*>
Linux: Requests the JVM to allocate the Java heap with large pages. If large pages are not available, the JVM does not start, displaying the error message `GC: system configuration does not support option --> '-Xlp'`. The JVM uses shmget() to allocate large pages for the heap. Large pages are supported by systems running Linux kernels v2.6 or higher. By default, large pages are not used.

AIX, Linux, and Windows only: If a <*size*> is specified, the JVM attempts to allocate the JIT code cache memory using pages of that size. If unsuccessful, or if executable pages of that size are not supported, the JIT code cache memory is allocated using the smallest available executable page size.

-Xmso<*size*>
Sets the initial stack size for operating system threads. The default value can be determined by running the command:

```
java -verbose:sizes
```

The maximum value for the stack size varies according to platform and specific machine configuration. If you exceed the maximum value, a `java/lang/OutOfMemoryError` message is reported.

-Xnoagent
Disables support for the old JDB debugger.

-Xnoclassgc
Disables dynamic class unloading. This option disables the release of native and Java heap storage associated with Java class loaders and classes that are no longer being used by the JVM. The default behavior is as defined by **-Xclassgc**. Enabling this option is not recommended except under the direction of the IBM Java support team. The reason is the option can cause unlimited native memory growth, leading to out-of-memory errors.

-Xnolinenumbers
Disables the line numbers for debugging. See also **-Xlinenumbers**. By default, line number are on.

-Xnosigcatch
Disables JVM signal handling code. See also **-Xsigcatch**. By default, signal handling is enabled.

-Xnosigchain
Disables signal handler chaining. See also **-Xsigchain**. By default, the signal handler chaining is enabled.

-Xoptionsfile=<*file*>

Specifies a file that contains JVM options and definitions. By default, no option file is used.

The options file does not support these options:

- **-version**
- **-showversion**
- **-fullversion**
- **-Xjarversion**
- **-memorycheck**
- **-assert**
- **-help**

<*file*> contains options that are processed as if they had been entered directly as command-line options. For example, the options file might contain:

```
-DuserString=ABC123
-Xmx256MB
```

Some options use quoted strings as parameters. Do not split quoted strings over multiple lines using the line continuation character '\'. The '¥' character is not supported as a line continuation character. For example, the following example is not valid in an options file:

```
-Xevents=vmstop,exec="cmd /c \
echo %pid has finished."
```

The following example is valid in an options file:

```
-Xevents=vmstop, \
exec="cmd /c echo %pid has finished."
```

-Xoss<*size*>
Recognized but deprecated. Use **-Xss** and **-Xmso**. Sets the maximum Java stack size for any thread. The maximum value for the stack size varies according to platform and specific machine configuration. If you exceed the maximum value, a `java/lang/OutOfMemoryError` message is reported.

-Xrdbginfo:<*host*>:<*port*>
Loads the remote debug information server with the specified host and port. By default, the remote debug information server is disabled.

-Xrs
Disables signal handling in the JVM. Setting **-Xrs** prevents the Java runtime from handling any internally or externally generated signals such as SIGSEGV and SIGABRT. Any signals raised are handled by the default operating system handlers.

-Xrun<*library name*>**[:**<*options*>**]**
Loads helper libraries. To load multiple libraries, specify it more than once on the command line. Examples of these libraries are:

-Xrunhprof[:help] | [:<*option*>**=**<*value*>**, ...]**
Performs heap, CPU, or monitor profiling.

-Xrunjdwp[:help] | [:<*option*>=<*value*>, ...]
Loads debugging libraries to support the remote debugging of applications. This option is the same as **-Xdbg**.

-Xrunjnichk[:help] | [:<*option*>=<*value*>, ...]
Deprecated. Use **-Xcheck:jni** instead.

-Xscmx<*size*>
Specifies cache size. This option applies only if a cache is being created and no cache of the same name exists. The default cache size is platform-dependent. You can find out the size value being used by adding **-verbose:sizes** as a command-line argument. Minimum cache size is 4 KB. Maximum cache size is platform-dependent. The size of cache that you can specify is limited by the amount of physical memory and paging space available to the system. The virtual address space of a process is shared between the shared classes cache and the Java heap. Increasing the maximum size of the Java heap reduces the size of the shared classes cache that you can create.

-XselectiveDebug
Enables selective debugging. Use the com.ibm.jvm.Debuggable annotation to mark classes and methods that must be available for debugging. The JVM optimizes methods that do not need debugging to provide better performance in a debugging environment. See the *User Guide* for your platform for more information.

-Xshareclasses:<*suboptions*>

Enables class sharing. This option can take a number of suboptions, some of which are cache utilities. Cache utilities perform the required operation on the specified cache, without starting the VM. You can combine multiple suboptions, separated by commas, but the cache utilities are mutually exclusive.

You can use the following suboptions with the **-Xshareclasses** option:

cacheDir=<*directory*>
Sets the directory in which cache data is read and written. By default, <*directory*> is `/tmp/javasharedresources` on Linux, AIX, z/OS, and IBM i. You must have sufficient permissions in <*directory*>. The JVM writes persistent cache files directly into the directory specified. Persistent cache files can be safely moved and deleted from the file system. Nonpersistent caches are stored in shared memory and have control files that describe the location of the memory. Control files are stored in a `javasharedresources` subdirectory of the **cacheDir** specified. Do not move or delete control files in this directory. The **listAllCaches** utility, the **destroyAll** utility, and the **expire** suboption work only in the scope of a given **cacheDir**.

cacheRetransformed
Enables caching of classes that have been transformed using the JVMTI RetransformClasses function. See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html) section on the "JVMTI redefinition and retransformation of classes" for more information.

destroy (Utility option)
Destroys a cache specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. A cache can be destroyed only if all JVMs using it have shut down and the user has sufficient permissions.

destroyAll (Utility option)
Tries to destroy all caches available using the specified **cacheDir** and

nonpersistent suboptions. A cache can be destroyed only if all JVMs using it have shut down and the user has sufficient permissions.

expire=<*time in minutes*> **(Utility option)**
Destroys all caches that have been unused for the time specified before loading shared classes. This option is not a utility option because it does not cause the JVM to exit.

groupAccess
Sets operating system permissions on a new cache to allow group access to the cache. The default is user access only.

help
Lists all the command-line options.

listAllCaches (Utility option)
Lists all the compatible and incompatible caches that exist in the specified cache directory. If you do not specify **cacheDir**, the default directory is used. Summary information, such as Java version and current usage, is displayed for each cache.

mprotect=[all | default | none]
By default, the memory pages containing the cache are protected at all times, unless a specific page is being updated. This protection helps prevent accidental or deliberate corruption to the cache. The cache header is not protected by default because this protection has a small performance cost. Specifying `all` ensures that all the cache pages are protected, including the header. Specifying `none` disables the page protection.

modified=<*modified context*>
Used when a JVMTI agent is installed that might modify bytecode at run time. If you do not specify this suboption and a bytecode modification agent is installed, classes are safely shared with an extra performance cost. The <*modified context*> is a descriptor chosen by the user; for example, *myModification1*. This option partitions the cache, so that only JVMs using context *myModification1* can share the same classes. For instance, if you run an application with a modification context and then run it again with a different modification context, all classes are stored twice in the cache. See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html) section “Dealing with runtime bytecode modification” for more information.

name=<*name*>
Connects to a cache of a given name, creating the cache if it does not exist. This option is also used to indicate the cache that is to be modified by cache utilities; for example, **destroy**. Use the **listAllCaches** utility to show which named caches are currently available. If you do not specify a name, the default name “sharedcc_%u” is used. "%u" in the cache name inserts the current user name. You can specify “%g” in the cache name to insert the current group name.

noaot
Disables caching and loading of AOT code.

noBootclasspath
Disables the storage of classes loaded by the bootstrap class loader in the shared classes cache. Often used with the SharedClassURLFilter API to control exactly which classes are cached. See the Diagnostics Guide

(http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html) section on the “SharedClassHelper API” for more information about shared class filtering.

none
Added to the end of a command line, disables class data sharing. This suboption overrides class sharing arguments found earlier on the command line.

nonfatal
Allows the JVM to start even if class data sharing fails. Normal behavior for the JVM is to refuse to start if class data sharing fails. If you select **nonfatal** and the shared classes cache fails to initialize, the JVM attempts to connect to the cache in read-only mode. If this attempt fails, the JVM starts without class data sharing.

nonpersistent
Uses a nonpersistent cache. The cache is lost when the operating system shuts down. Nonpersistent and persistent caches can have the same name. You must always use the **nonpersistent** suboption when running utilities such as **destroy** on a nonpersistent cache.

persistent (default for Windows and Linux platforms)
Uses a persistent cache. The cache is created on disk, which persists beyond operating system restarts. Nonpersistent and persistent caches can have the same name.

printAllStats (Utility option)
Displays detailed information about the contents of the cache specified in the **name=**<*name*> suboption. If the name is not specified, statistics are displayed about the default cache. Every class is listed in chronological order with a reference to the location from which it was loaded. See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html) section on the “printAllStats utility” for more information.

printStats (Utility option)
Displays summary information for the cache specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. The most useful information displayed is how full the cache is and how many classes it contains. Stale classes are classes that have been updated on the file system and which the cache has therefore marked "stale". Stale classes are not purged from the cache and can be reused. See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html) section on the “printStats utility” for more information.

readonly
Opens an existing cache with read-only permissions. The JVM does not create a new cache with this suboption. Opening a cache read-only prevents the JVM from making any updates to the cache. It also allows the JVM to connect to caches created by other users or groups without requiring write access. By default, this suboption is not specified.

reset
Causes a cache to be destroyed and then re-created when the JVM starts up. This option can be added to the end of a command line as **-Xshareclasses:reset**.

safemode
Forces the JVM to load all classes from disk and apply the modifications to

those classes (if applicable). See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html) section on “Using the safemode option” for more information.

silent
Disables all shared class messages, including error messages. Unrecoverable error messages, which prevent the JVM from initializing, are displayed.

verbose
Gives detailed output on the cache I/O activity, listing information about classes being stored and found. Each class loader is given a unique ID (the bootstrap loader is always 0) and the output shows the class loader hierarchy at work, where class loaders must ask their parents for a class before they can load it themselves. It is typical to see many failed requests; this behavior is expected for the class loader hierarchy. The standard option **-verbose:class** also enables class sharing verbose output if class sharing is enabled.

verboseAOT
Enables verbose output when compiled AOT code is being found or stored in the cache. AOT code is generated heuristically. You might not see any AOT code generated at all for a small application. You can disable AOT caching using the **noaot** suboption. See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html) section on “JITM messages” for a list of the messages produced.

verboseHelper
Enables verbose output for the Java Helper API. This output shows you how the Helper API is used by your ClassLoader.

verboseIO
Gives detailed output on the cache I/O activity, listing information about classes being stored and found. Each class loader is given a unique ID (the bootstrap loader is always 0) and the output shows the class loader hierarchy at work, where class loaders must ask their parents for a class before they can load it themselves. It is typical to see many failed requests; this behavior is expected for the class loader hierarchy.

-Xsigcatch
Enables VM signal handling code. See also **-Xnosigcatch**. By default, signal handling is enabled.

-Xsigchain
Enables signal handler chaining. See also **-Xnosigchain**. By default, signal handler chaining is enabled.

-Xss<*size*>
Sets the maximum stack size for Java threads. The default is 256 KB for 32-bit JVMs and 512 KB for 64-bit JVMs. The maximum value varies according to platform and specific machine configuration. If you exceed the maximum value, a `java/lang/OutOfMemoryError` message is reported.

-Xssi<*size*>
Sets the stack size increment for Java threads. When the stack for a Java thread becomes full it is increased in size by this value until the maximum size (**-Xss**) is reached. The default is 16 KB.

-Xthr:minimizeUserCPU
Minimizes user-mode CPU usage in thread synchronization where possible. The reduction in CPU usage might be a trade-off in exchange for lower performance.

-Xtrace[:help] | [:<*option*>=<*value*>, ...]
See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html) section on the "Controlling the trace" for more information.

-Xverify[:<*option*>**]**
With no parameters, enables the verifier, which is the default. Therefore, if used on its own with no parameters, for example, **-Xverify**, this option does nothing. Optional parameters are as follows:

- **all** - enable maximum verification
- **none** - disable the verifier
- **remote** - enables strict class-loading checks on remotely loaded classes

The verifier is on by default and must be enabled for all production servers. Running with the verifier off is not a supported configuration. If you encounter problems and the verifier was turned off using **-Xverify:none**, remove this option and try to reproduce the problem.

-Xzero[:<*option*>**]**

Enables reduction of the memory footprint of Java when concurrently running multiple Java invocations. **-Xzero** might not be appropriate for all types of applications because it changes the implementation of `java.util.ZipFile`, which might cause extra memory usage. **-Xzero** includes the optional parameters:

- **j9zip** - enables the j9zip suboption
- **noj9zip** - disables the j9zip suboption
- **sharezip** - enables the sharezip suboption
- **nosharezip** - disables the sharezip suboption
- **none** - disables all suboptions
- **describe** - prints the suboptions in effect

Because future versions might include more default options, **-Xzero** options are used to specify the suboptions that you want to disable. By default, **-Xzero** enables **j9zip** and **sharezip**. A combination of **j9zip** and **sharezip** enables all jar files to have shared caches:

- **j9zip** - uses a new `java.util.ZipFile` implementation. This suboption is not a requirement for **sharezip**; however, if **j9zip** is not enabled, only the bootstrap jars have shared caches.
- **sharezip** - puts the j9zip cache into shared memory. The j9zip cache is a map of names to file positions used by the compression implementation to quickly find compressed entries. You must enable **-Xshareclasses** to avoid a warning message. When using the **sharezip** suboption, note that this suboption allows every opened .zip file and .jar file to store the j9zip cache in shared memory, so you might fill the shared memory when opening multiple new .zip files and .jar files. The affected API is `java.util.zip.ZipFile` (superclass of `java.util.jar.JarFile`). The .zip and .jar files do not have to be on a class path.

-Xzero is available only in Java 6 SR1 and beyond. When enabled, the system property **com.ibm.zero.version** is defined, and has a current value of 1. For Java 6 SR1 and Java 6 SR2, the **-Xzero** option is accepted only on Windows

x86-32 and Linux x86-32 platforms. From Java 6 SR3, **-Xzero** is accepted on all platforms; however, it is enabled only on Windows x86-32 and Linux x86-32 platforms.

-XX command-line options:

JVM command-line options that are specified with -XX are not stable and are not recommended for casual use.

These options are subject to change without notice.

-XXallowvmshutdown:[false | true]
This option is provided as a workaround for customer applications that cannot shut down cleanly, as described in APAR IZ59734. Customers who need this workaround should use **-XXallowvmshutdown:false**. The default option is **-XXallowvmshutdown:true** forJava 6 SR5 onwards.

-XX:MaxDirectMemorySize=<*size*>
Sets the maximum size for an nio direct buffer. By default, the maximum size is 64 MB.

JIT and AOT command-line options

Use these JIT and AOT compiler command-line options to control code compilation.

You might need to read the section on JIT and AOT problems in the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html) to understand some of the references that are given here.

-Xaot[:<*parameter*>**=**<*value*>**, ...]**
With no parameters, enables the AOT compiler. The AOT compiler is enabled by default but is not active unless shared classes are enabled. Using this option on its own has no effect. Use this option to control the behavior of the AOT compiler. These parameters are useful:

count=<*n*>
Where <n> is the number of times a method is called before it is compiled. For example, setting `count=0` forces the AOT compiler to compile everything on first execution.

limitFile=(<*filename*>,<*m*>,<*n*>)
Compile only the methods listed on lines <*m*> to <*n*> in the specified limit file. Methods not listed in the limit file and methods listed on lines outside the range are not compiled.

loadExclude=<*methods*>
Do not load methods beginning with <*methods*>.

loadLimit=<*methods*>
Load methods beginning with <*methods*> only.

loadLimitFile=(<*filename*>,<*m*>,<*n*>)
Compile only the methods listed on lines <*m*> to <*n*> in the specified limit file. Methods not listed in the limit file and methods listed on lines outside the range are not compiled.

verbose
Displays information about the AOT and JIT compiler configuration and method compilation.

-Xcodecache<*size*>
Sets the size of each block of memory that is allocated to store native code of compiled Java methods. By default, this size is selected internally according to the CPU architecture and the capability of your system. If profiling tools show significant costs in trampolines (JVMTI identifies trampolines in a methodLoad2 event), that is a good prompt to change the size until the costs are reduced. Changing the size does not mean always increasing the size. The option provides the mechanism to tune for the right size until hot interblock calls are eliminated. A reasonable starting point to tune for the optimal size is (totalNumberByteOfCompiledMethods * 1.1). This option is used to tune performance.

-Xint
Makes the JVM use the Interpreter only, disabling the Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilers. By default, the JIT compiler is enabled. By default, the AOT compiler is enabled, but is not used by the JVM unless shared classes are also enabled.

-Xjit[:<*parameter*>**=**<*value*>**, ...]**
With no parameters, enables the JIT compiler. The JIT compiler is enabled by default, so using this option on its own has no effect. Use this option to control the behavior of the JIT compiler. Useful parameters are:

count=<*n*>
Where <n> is the number of times a method is called before it is compiled. For example, setting `count=0` forces the JIT compiler to compile everything on first execution.

limitFile=(<*filename*>**,** <*m*>**,** <*n*>**)**
Compile only the methods listed on lines <*m*> to <*n*> in the specified limit file. Methods not listed in the limit file and methods listed on lines outside the range are not compiled.

optlevel=[noOpt | cold | warm | hot | veryHot | scorching]
Forces the JIT compiler to compile all methods at a specific optimization level. Specifying **optlevel** might have an unexpected effect on performance, including lower overall performance.

verbose
Displays information about the JIT and AOT compiler configuration and method compilation.

-Xnoaot
Turns off the AOT compiler and disables the use of AOT-compiled code. By default, the AOT compiler is enabled but is active only when shared classes are also enabled. Using this option does not affect the JIT compiler.

-Xnojit
Turns off the JIT compiler. By default, the JIT compiler is enabled. This option does not affect the AOT compiler. `java -Xnojit -version` displays `JIT enabled` if the AOT compiler is enabled.

-Xquickstart
Causes the JIT compiler to run with a subset of optimizations. The effect is faster compilation times that improve startup time, but longer running applications might run slower. When the AOT compiler is active (both shared classes and AOT compilation enabled), **-Xquickstart** causes all methods to be AOT compiled. The AOT compilation improves the startup time of subsequent runs, but might reduce performance for longer running applications. **-Xquickstart** can degrade performance if it is used with long-running

applications that contain hot methods. The implementation of **-Xquickstart** is subject to change in future releases. By default, **-Xquickstart** is disabled.

–XsamplingExpirationTime<*time*>
Disables the JIT sampling thread after <*time*> seconds. When the JIT sampling thread is disabled, no CPU cycles are consumed by an idle JVM.

-Xscmaxaot<*size*>
Optionally applies a maximum number of bytes in the class cache that can be used for AOT data. This option is useful if you want a certain amount of cache space guaranteed for non-AOT data. If this option is not specified, the maximum limit for AOT data is the amount of free space in the cache. The value of this option must not be smaller than the value of **-Xscminaot** and must not be larger than the value of **-Xscmx**.

-Xscminaot<*size*>
Optionally applies a minimum number of bytes in the class cache to reserve for AOT data. If this option is not specified, no space is reserved for AOT data, although AOT data is still written to the cache until the cache is full or the **-Xscmaxaot** limit is reached. The value of this option must not exceed the value of **-Xscmx** or **-Xscmaxaot**. The value of **-Xscminaot** must always be considerably less than the total cache size, because AOT data can be created only for cached classes. If the value of **-Xscminaot** equals the value of **-Xscmx**, no class data or AOT data can be stored.

Garbage Collector command-line options

Use these Garbage Collector command-line options to control garbage collection.

You might need to read the section on “Memory management” in the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html) to understand some of the references that are given here.

The **-verbose:gc** option detailed in the section on “-verbose:gc logging” in the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html) is the main diagnostic aid that is available for runtime analysis of the Garbage Collector. However, additional command-line options are available that affect the behavior of the Garbage Collector and might aid diagnostics.

For options that take a <*size*> parameter, suffix the number with "k" or "K" to indicate kilobytes, "m" or "M" to indicate megabytes, or "g" or "G" to indicate gigabytes.

For options that take a <*percentage*> parameter, use a number from 0 to 1, for example, 50% is 0.5.

-Xalwaysclassgc
Always perform dynamic class unloading checks during global collection. The default behavior is as defined by **-Xclassgc**.

-Xclassgc
Enables the collection of class objects only on class loader changes. This behavior is the default.

-Xcompactexplicitgc
Enables full compaction each time System.gc() is called.

-Xcompactgc
Compacts on all garbage collections (system and global).

The default (no compaction option specified) makes the GC compact based on a series of triggers that attempt to compact only when it is beneficial to the future performance of the JVM.

-Xconcurrentbackground<*number*>
Specifies the number of low-priority background threads attached to assist the mutator threads in concurrent mark. The default is 0 on Linux zSeries and 1 on all other platforms.

-Xconcurrentlevel<*number*>
Specifies the allocation "tax" rate. It indicates the ratio between the amount of heap allocated and the amount of heap marked. The default is 8.

-Xconcurrentslack<*size*>
Attempts to keep the specified amount of the heap space free in concurrent collectors by starting the concurrent operations earlier. This behavior can sometimes alleviate pause time problems in concurrent collectors at the cost of longer concurrent cycles, affecting total throughput. The default is 0, which is optimal for most applications.

-Xconmeter:<*soa* | *loa* | *dynamic*>
This option determines the usage of which area, LOA (Large Object Area) or SOA (Small Object Area), is metered and hence which allocations are taxed during concurrent mark. Using **-Xconmeter:soa** (the default) applies the allocation tax to allocations from the small object area (SOA). Using **-Xconmeter:loa** applies the allocation tax to allocations from the large object area (LOA). If **-Xconmeter:dynamic** is specified, the collector dynamically determines which area to meter based on which area is exhausted first, whether it is the SOA or the LOA.

-Xdisableexcessivegc
Disables the throwing of an OutOfMemory exception if excessive time is spent in the GC.

-Xdisableexplicitgc

Disables System.gc() calls.

Many applications still make an excessive number of explicit calls to System.gc() to request garbage collection. In many cases, these calls degrade performance through premature garbage collection and compactions. However, you cannot always remove the calls from the application.

The **-Xdisableexplicitgc** parameter allows the JVM to ignore these garbage collection suggestions. Typically, system administrators use this parameter in applications that show some benefit from its use.

By default, calls to System.gc() trigger a garbage collection.

-Xdisablestringconstantgc
Prevents strings in the string intern table from being collected.

-Xenableexcessivegc
If excessive time is spent in the GC, the option returns null for an allocate request and thus causes an OutOfMemory exception to be thrown. This action occurs only when the heap has been fully expanded and the time spent is making up at least 95%. This behavior is the default.

-Xenablestringconstantgc
Enables strings from the string intern table to be collected. This behavior is the default.

-Xgc:<*options*>
Passes options such as verbose, compact, and nocompact to the Garbage Collector.

-Xgc:splitheap
Allocates the new and old areas of the generational Java heap in separate areas of memory. Using a split heap forces the Garbage Collector to use the gencon policy and disables resizing of the new and old memory areas. See the section on the "Split heap" in the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html) for more information. By default, the Java heap is allocated in a single contiguous area of memory.

-Xgcpolicy:<optthruput | optavgpause | gencon | subpool (AIX, Linux and IBM i on IBM POWER® architecture, Linux and z/OS on zSeries) >
Controls the behavior of the Garbage Collector.

The optthruput option is the default and delivers high throughput to applications, but at the cost of occasional pauses. Disables concurrent mark.

The optavgpause option reduces the time that is spent in these garbage collection pauses and limits the effect of increasing heap size on the length of the garbage collection pause. Use optavgpause if your configuration has a large heap. Enables concurrent mark.

The gencon option requests the combined use of concurrent and generational GC to help minimize the time that is spent in any garbage collection pause.

The subpool option (AIX, Linux and IBM i on IBM POWER architecture, and z/OS) uses an improved object allocation algorithm to achieve better performance when allocating objects on the heap. This option might improve performance on large SMP systems.

-Xgcthreads<*number*>
Sets the number of threads that the Garbage Collector uses for parallel operations. This total number of GC threads is composed of one application thread with the remainder being dedicated GC threads. By default, the number is set to the number of physical CPUs present. To set it to a different number (for example 4), use **-Xgcthreads4**. The minimum valid value is 1, which disables parallel operations, at the cost of performance. No advantage is gained if you increase the number of threads above the default setting; you are recommended not to do so.

On systems running multiple JVMs or in LPAR environments where multiple JVMs can share the same physical CPUs, you might want to restrict the number of GC threads used by each JVM. The restriction helps prevent the total number of parallel operation GC threads for all JVMs exceeding the number of physical CPUs present, when multiple JVMs perform garbage collection at the same time.

-Xgcworkpackets<*number*>
Specifies the total number of work packets available in the global collector. If not specified, the collector allocates a number of packets based on the maximum heap size.

-Xloa
Allocates a large object area (LOA). Objects are allocated in this LOA rather than the SOA. By default, the LOA is enabled for all GC policies except for subpool, where the LOA is not available.

-Xloainitial<*percentage*>
Specifies the initial percentage (between 0 and 0.95) of the current tenure space allocated to the large object area (LOA). The default value is 0.05, which is 5%.

-Xloamaximum<*percentage*>
Specifies the maximum percentage (between 0 and 0.95) of the current tenure space allocated to the large object area (LOA). The default value is 0.5, which is 50%.

-Xloaminimum<*percentage*>
Specifies the minimum percentage (between 0 and 0.95) of the current tenure space allocated to the large object area (LOA). The LOA does not shrink below this value. The default value is 0, which is 0%.

-Xmaxe<*size*>
Sets the maximum amount by which the garbage collector expands the heap. Typically, the garbage collector expands the heap when the amount of free space falls below 30% (or by the amount specified using **-Xminf**), by the amount required to restore the free space to 30%. The **-Xmaxe** option limits the expansion to the specified value; for example `-Xmaxe10M` limits the expansion to 10 MB. By default, there is no maximum expansion size.

-Xmaxf<*percentage*>
Specifies the maximum percentage of heap that must be free after a garbage collection. If the free space exceeds this amount, the JVM tries to shrink the heap. The default value is 0.6 (60%).

-Xmaxt<*percentage*>
Specifies the maximum percentage of time to be spent in Garbage Collection. If the percentage of time rises above this value, the JVM tries to expand the heap. The default value is 13%.

-Xmca<*size*>
Sets the expansion step for the memory allocated to store the RAM portion of loaded classes. Each time more memory is required to store classes in RAM, the allocated memory is increased by this amount. By default, the expansion step is 32 KB. Use the **-verbose:sizes** option to determine the value that the VM is using. If the expansion step size you choose is too large, `OutOfMemoryError` is reported. The exact value of a "too large" expansion step size varies according to the platform and the specific machine configuration.

-Xmco<*size*>
Sets the expansion step for the memory allocated to store the ROM portion of loaded classes. Each time more memory is required to store classes in ROM, the allocated memory is increased by this amount. By default, the expansion step is 128 KB. Use the **-verbose:sizes** option to determine the value that the VM is using. If the expansion step size you choose is too large, `OutOfMemoryError` is reported. The exact value of a "too large" expansion step size varies according to the platform and the specific machine configuration.

-Xmine<*size*>
Sets the minimum amount by which the Garbage Collector expands the heap. Typically, the garbage collector expands the heap by the amount required to restore the free space to 30% (or the amount specified using **-Xminf**). The **-Xmine** option sets the expansion to be at least the specified value; for example, `-Xmine50M` sets the expansion size to a minimum of 50 MB. By default, the minimum expansion size is 1 MB.

Note: Do not use this option with the metronome garbage collector (-Xrealtime).

-Xminf<*percentage*>
Specifies the minimum percentage of heap to be left free after a garbage collection. If the free space falls below this amount, the JVM attempts to expand the heap. The default value is 30%.

-Xmint<*percentage*>
Specifies the minimum percentage of time to spend in Garbage Collection. If the percentage of time drops below this value, the JVM tries to shrink the heap. The default value is 5%.

-Xmn<*size*>
Sets the initial and maximum size of the new area to the specified value when using **-Xgcpolicy:gencon**. Equivalent to setting both **-Xmns** and **-Xmnx**. If you set either **-Xmns** or **-Xmnx**, you cannot set **-Xmn**. If you try to set **-Xmn** with either **-Xmns** or **-Xmnx**, the VM does not start, returning an error. By default, **-Xmn** is not set. If the scavenger is disabled, this option is ignored.

-Xmns<*size*>
Sets the initial size of the new area to the specified value when using **-Xgcpolicy:gencon**. By default, this option is set to 25% of the value of the **-Xms** option. This option returns an error if you try to use it with **-Xmn**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using. If the scavenger is disabled, this option is ignored.

-Xmnx<*size*>
Sets the maximum size of the new area to the specified value when using **-Xgcpolicy:gencon**. By default, this option is set to 25% of the value of the **-Xmx** option. This option returns an error if you try to use it with **-Xmn**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using. If the scavenger is disabled, this option is ignored.

-Xmo<*size*>
Sets the initial and maximum size of the old (tenured) heap to the specified value when using **-Xgcpolicy:gencon**. Equivalent to setting both **-Xmos** and **-Xmox**. If you set either **-Xmos** or **-Xmox**, you cannot set **-Xmo**. If you try to set **-Xmo** with either **-Xmos** or **-Xmox**, the VM does not start, returning an error. By default, **-Xmo** is not set.

-Xmoi<*size*>
Sets the amount the Java heap is incremented when using **-Xgcpolicy:gencon**. If set to zero, no expansion is allowed. By default, the increment size is calculated on the expansion size, set by **-Xmine** and **-Xminf**.

-Xmos<*size*>
Sets the initial size of the old (tenure) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is set to 75% of the value of the **-Xms** option. This option returns an error if you try to use it with **-Xmo**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

-Xmox<*size*>
Sets the maximum size of the old (tenure) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is set to the same value as the **-Xmx** option. This option returns an error if you try to use it with **-Xmo**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

-Xmr<*size*>
Sets the size of the Garbage Collection "remembered set". This set is a list of

objects in the old (tenured) heap that have references to objects in the new area. By default, this option is set to 16 K.

-Xmrx<*size*>
Sets the remembered maximum size setting.

-Xms<*size*>
Sets the initial Java heap size. You can also use the **-Xmo** option. The minimum size is 1 MB.

If scavenger is enabled, **-Xms** >= **-Xmn** + **-Xmo**.

If scavenger is disabled, **-Xms** >= **-Xmo**.

-Xmx<*size*>
Sets the maximum memory size (**-Xmx** >= **-Xms**)

Examples of the use of **-Xms** and **-Xmx**:

-Xms2m -Xmx64m
Heap starts at 2 MB and grows to a maximum of 64 MB.

-Xms100m -Xmx100m
Heap starts at 100 MB and never grows.

-Xms20m -Xmx1024m
Heap starts at 20 MB and grows to a maximum of 1 GB.

-Xms50m
Heap starts at 50 MB and grows to the default maximum.

-Xmx256m
Heap starts at default initial value and grows to a maximum of 256 MB.

-Xnoclassgc
Disables class garbage collection. This option switches off garbage collection of storage associated with Java classes that are no longer being used by the JVM. The default behavior is as defined by **-Xclassgc**. By default, class garbage collection is performed.

-Xnocompactexplicitgc
Disables compaction on System.gc() calls. Compaction takes place on global garbage collections if you specify **-Xcompactgc** or if compaction triggers are met. By default, compaction is enabled on calls to System.gc().

-Xnocompactgc
Disables compaction on all garbage collections (system or global). By default, compaction is enabled.

-Xnoloa
Prevents allocation of a large object area; all objects are allocated in the SOA. See also **-Xloa**.

-Xnopartialcompactgc
Disables incremental compaction. See also **-Xpartialcompactgc**.

-Xpartialcompactgc
Enables incremental compaction. See also **-Xnopartialcompactgc**. By default, this option is not set, so all compactions are full.

-Xsoftmx<*size*> **(AIX only)**
This option sets the initial maximum size of the Java heap. Use the **-Xmx** option to set the maximum heap size. Use the AIX DLPAR API in your application to alter the heap size limit between **-Xms** and **-Xmx** at run time. By default, this option is set to the same value as **-Xmx**.

-Xsoftrefthreshold<*number*>
Sets the value used by the GC to determine the number of GCs after which a soft reference is cleared if its referent has not been marked. The default is 32, meaning that the soft reference is cleared after 32 * (percentage of free heap space) GC cycles where its referent was not marked.

-Xtgc:<*arguments*>
Provides GC tracing options, where <*arguments*> is a comma-separated list containing one or more of the following arguments:

backtrace
Before a garbage collection, a single line is printed containing the name of the master thread for garbage collection, as well as the value of the osThread slot in the J9VMThread structure.

compaction
Prints extra information showing the relative time spent by threads in the “move” and “fixup” phases of compaction

concurrent
Prints extra information showing the activity of the concurrent mark background thread

dump
Prints a line of output for every free chunk of memory in the system, including "dark matter" (free chunks that are not on the free list for some reason, typically because they are too small). Each line contains the base address and the size in bytes of the chunk. If the chunk is followed in the heap by an object, the size and class name of the object is also printed. This argument has a similar effect to the **terse** argument.

freeList
Before a garbage collection, prints information about the free list and allocation statistics since the last GC. Prints the number of items on the free list, including "deferred" entries (with the scavenger, the unused space is a deferred free list entry). For TLH and non-TLH allocations, prints the total number of allocations, the average allocation size, and the total number of bytes discarded during allocation. For non-TLH allocations, also included is the average number of entries that were searched before a sufficiently large entry was found.

parallel
Produces statistics on the activity of the parallel threads during the mark and sweep phases of a global GC.

references
Prints extra information every time that a reference object is enqueued for finalization, showing the reference type, reference address, and referent address.

scavenger
Prints extra information after each scavenger collection. A histogram is produced showing the number of instances of each class, and their relative ages, present in the survivor space. The information is obtained by performing a linear walk-through of the space.

terse
Dumps the contents of the entire heap before and after a garbage collection. For each object or free chunk in the heap, a line of trace output

is produced. Each line contains the base address, "a" if it is an allocated object, and "f" if it is a free chunk, the size of the chunk in bytes, and, if it is an object, its class name.

-Xverbosegclog[:<*file*>[,<X>,<Y>]]

Causes **-verbose:gc** output to be written to the specified file. If the file cannot be found, **-verbose:gc** tries to create the file, and then continues as normal if it is successful. If it cannot create the file (for example, if an invalid filename is passed into the command), it redirects the output to stderr.

If you specify <*X*> and <*Y*> the **-verbose:gc** output is redirected to X files, each containing Y GC cycles.

The dump agent tokens can be used in the filename. See the Diagnostics Guide (http://www.ibm.com/developerworks/java/jdk/diagnosis/60.html) section on the "Dump agent tokens" for more information. If you do not specify <*file*>, `verbosegc.%Y%m%d.%H%M%S.%pid.txt` is used.

By default, no verbose GC logging occurs.

Default settings for the JVM

This appendix shows the default settings that the JVM uses (how the JVM operates if you do not apply any changes to its environment). The tables show the JVM operation and the default setting.

The last column shows how the operation setting is affected and is set as follows:

- **e** - setting controlled by environment variable only
- **c** - setting controlled by command-line parameter only
- **ec** - setting controlled by both (command-line always takes precedence) All the settings are described elsewhere in this document. These tables are only a quick reference to the JVM vanilla state

For default GC settings, see the following table:

JVM setting	Default	Setting affected by
Javadumps	Enabled	ec
Javadumps on out of memory	Enabled	ec
Heapdumps	Disabled	ec
Heapdumps on out of memory	Enabled	ec
Sysdumps	Enabled	ec
Where dump files are produced	Current directory	ec
Verbose output	Disabled	c
Boot classpath search	Disabled	c
JNI checks	Disabled	c
Remote debugging	Disabled	c
Strict conformancy checks	Disabled	c
Quickstart	Disabled	c
Remote debug info server	Disabled	c
Reduced signalling	Disabled	c
Signal handler chaining	Enabled	c

JVM setting	Default	Setting affected by
Classpath	Not set	ec
Class data sharing	Disabled	c
Accessibility support	Enabled	e
JIT compiler	Enabled	ec
AOT compiler (AOT is not used by the JVM unless shared classes are also enabled)	Enabled	c
JIT debug options	Disabled	c
Java2D max size of fonts with algorithmic bold	14 point	e
Java2D use rendered bitmaps in scalable fonts	Enabled	e
Java2D freetype font rasterizing	Enabled	e
Java2D use AWT fonts	Disabled	e

JVM setting	AIX	IBM i	Linux	Windows	z/OS	Setting affected by
Default locale	None	None	None	N/A	None	e
Time to wait before starting plug-in	N/A	N/A	Zero	N/A	N/A	e
Temporary directory	/tmp	/tmp	/tmp	c:\temp	/tmp	e
Plug-in redirection	None	None	None	N/A	None	e
IM switching	Disabled	Disabled	Disabled	N/A	Disabled	e
IM modifiers	Disabled	Disabled	Disabled	N/A	Disabled	e
Thread model	N/A	N/A	N/A	N/A	Native	e
Initial stack size for Java Threads 32-bit. Use: **-Xiss<size>**	2 KB	2 KB	2 KB	2 KB	2 KB	c
Maximum stack size for Java Threads 32-bit. Use: **-Xss<size>**	256 KB	256 KB	256 KB	256 KB	256 KB	c
Stack size for OS Threads 32-bit. Use **-Xmso<size>**	256 KB	256 KB	256 KB	32 KB	256 KB	c
Initial stack size for Java Threads 64-bit. Use: **-Xiss<size>**	2 KB	N/A	2 KB	2 KB	2 KB	c
Maximum stack size for Java Threads 64-bit. Use: **-Xss<size>**	512 KB	N/A	512 KB	512 KB	512 KB	c
Stack size for OS Threads 64-bit. Use **-Xmso<size>**	256 KB	N/A	256 KB	256 KB	256 KB	c
Initial heap size. Use **-Xms<size>**	4 MB	4 MB	4 MB	4 MB	4 MB	c
Maximum Java heap size. Use **-Xmx<size>**	Half the available memory with a minimum of 16 MB and a maximum of 512 MB	2 GB	Half the available memory with a minimum of 16 MB and a maximum of 512 MB	Half the real memory with a minimum of 16 MB and a maximum of 2 GB	Half the available memory with a minimum of 16 MB and a maximum of 512 MB	c

"available memory" is the smallest of real (physical) memory and the **RLIMIT_AS** value.

Known limitations

Known limitations on IBM WebSphere Real Time for RT Linux.

CUPS support

IBM WebSphere Real Time for RT Linux does not support printing using the CUPS interface.

Calculation of the mean time to acquire exclusive access reports incorrect values

Verbose GC output now contains additional information to help diagnose problems. One of the new entries is the time spent quiescing threads in order to perform a garbage collection quanta, otherwise known as exclusive access time. There is a known bug with the calculation of the mean time to acquire exclusive access, which results in incorrect values being reported. The following code sample provides an example:

```
<gc type="heartbeat" id="6" timestamp="Jul 30 14:43:25 2007" intervalms="302.461">
<summary quantumcount="1">
<quantum minms="0.505" meanms="0.505" maxms="0.505" />
<exclusiveaccess minms="0.008" meanms="0.066" maxms="0.058" />
<heap minfree="32768" meanfree="32768" maxfree="32768" />
<immortal minfree="16054204" meanfree="16054204" maxfree="16054204" />
<gcthreadpriority max="11" min="11" />
</summary>
</gc>
```

As illustrated in the example, the mean time to quiesce threads is greater than the maximum time, which is incorrect. Do not trust the `meanms` value for exclusive access in any of the verbose GC stanzas output by the JVM.

JConsole monitoring tool Local tab

In the IBM JConsole tool, the **Local** tab, which allows you to connect to other Virtual Machines on the same system, is not available. Also, the corresponding command line **pid** option is not supported. Instead, use the **Remote** tab in JConsole to connect to the Virtual Machine that you want to monitor. Alternatively, use the **connection** command-line option, specifying a host of `localhost` and a port number. When you start the application that you want to monitor, set these command-line options:

-Dcom.sun.management.jmxremote.port=<*value*>
Specifies the port the management agent listens on.

-Dcom.sun.management.jmxremote.authenticate=false
Disables authentication unless you have created a user name file.

-Dcom.sun.management.jmxremote.ssl=false
Disables SSL encryption.

Incorrect stack traces when loading new classes after an Exception is caught

If new classes are loaded after an Exception has been caught, the stack trace contained in the Exception might become incorrect. The stack trace becomes

incorrect if classes in the stack trace are unloaded, and new classes are loaded into their memory segments.

Web Start and Java 1.3 applications

The WebSphere Real Time for RT Linux Web Start does not support launching Java 1.3 applications.

Slow DSA key pair generation

Creating DSA key pairs of unusual lengths can take a significant amount of time on slow machines. Do not interpret the delay as a stop or endless loop, because the process finishes if sufficient time is allowed. The DSA key generation algorithm has been optimized to generate standard key lengths (for instance, 512, 1024) more quickly than others.

Creating a JVM using JNI

Native programs cannot create a VM with JNI_VERSION_1_1(0x00010001) interfaces. You cannot call JNI_CreateJavaVM() and pass it a version of JNI_VERSION_1_1(0x00010001). The versions that can be passed are:

- JNI_VERSION_1_2(0x00010002)
- JNI_VERSION_1_4(0x00010004)

The VM created is determined by the Java libraries present (that is, 1.2.2, 1.3.x, 1.4.x, 5.x, 6.x), not the one that is implied by the JNI interface version passed.

The interface version does not affect any area of VM behavior other than the functions available to native code.

Window managers and keyboard shortcuts

Your window manager might override some of the Java keyboard shortcuts. If you need to use an overridden Java keyboard shortcut, consult your operating system manual and change your window manager keyboard shortcuts.

X Window System file descriptors

The X Window System is unable to use file descriptors above 255. Because the JVM holds file descriptors for open jar files, X can run out of file descriptors. As a workaround, you can set the **JAVA_HIGH_ZIPFDS** environment variable to tell the JVM to use higher file descriptors for jar files.

To use the **JAVA_HIGH_ZIPFDS** environment variable, set it to a value in the range 0 - 512. The JVM then opens the first jar files using file descriptors up to 1024. For example, if your program is likely to load 300 jar files:

```
export JAVA_HIGH_ZIPFDS=300
```

The first 300 jar files are then loaded using the file descriptors 724 - 1023. Any jar files opened after that are opened in the typical range.

DBCS and the KDE clipboard

You might not be able to use the system clipboard with double-byte character set (DBCS) to copy information between Linux applications and Java applications if you are running the K Desktop Environment (KDE).

ThreadMXBean Thread User CPU Time limitation

There is no way to distinguish between user mode CPU time and system mode CPU time on this platform. ThreadMXBean.getThreadUserTime(), ThreadMXBean.getThreadCpuTime(), ThreadMXBean.getCurrentThreadUserTime(), and ThreadMXBean.getCurrentThreadCpuTime() all return the total CPU time for the required thread.

KeyEvents and window managers

KeyEvent results that include the **Alt** key might differ between window managers in Linux. They also differ from results of other operating systems. When using the default settings, **Ctrl+Alt+A** in the KWin window manager produces a KeyEvent, whereas **Ctrl+Alt+A** in the Metacity window manager does not produce a key event.

The X Window System and the Meta key

On the Linux X Window System, different key codes are generated when certain keys are pressed at the same time. For example, the key code 64 is returned when you press Alt_L or Meta_L. Similarly, the key code 113 is returned when you press Alt_R or Meta_R. You can check the exact values by typing the following instruction at a shell prompt:

```
xmodmap -pk
```

With these default settings, WebSphere Real Time for RT Linux considers that the Meta and Alt keys are pressed together. As a workaround, remove the Meta_x mapping by typing the following instruction at a shell prompt:

```
xmodmap -e "keysym Alt_L = Alt_L" -e "keysym Alt_R = Alt_R"
```

This workaround might affect other X Window System applications that are running on the same display if they use the Meta-key that was removed.

SIGSEGV when creating a JVM using JNI

A call to JNI_CreateJavaVM() from a JNI application might cause a segmentation fault (signal SIGSEGV); to avoid this fault, rebuild your JNI program specifying the option **–lpthread**.

Lack of resources with highly threaded applications

If you are running with many concurrent threads, you might get a warning message:

```
java.lang.OutOfMemoryError
```

The message is an indication that your machine is running out of system resources and messages can be caused by the following reasons:

- If your Linux installation uses LinuxThreads, rather than NPTL, the number of processes created exceeds your user limit.

- Not enough system resources are available to create new threads. In this case, you might also see other Java exceptions, depending on what your application is running.
- Kernel memory is either running out or is fragmented. You can see corresponding Out of Memory kernel messages in `/var/log/messages`. The messages are associated with the ID of the killed process.

Try tuning your system to increase the corresponding system resources.

X Server and client font problems

When running a Java AWT or Swing application on a Linux machine and exporting the display to a second machine, you might experience problems displaying some dialogs if the set of fonts loaded on the X client machine is different from the set loaded on the X server machine. To avoid this problem, install the same fonts on both machines.

UTF-8 encoding and MalformedInputExceptions

If your system locale is using a UTF-8 encoding, some tools might throw a sun.io.MalformedInputException. To find out whether your system is using a UTF-8 encoding, examine the locale-specific environment variables such as **LANG** or **LC_ALL** to see if they end with the ".UTF-8" suffix. If you get the warning sun.io.MalformedInputException, change characters that are not in the 7-bit ASCII range (0x00 - 0x7f) and are not represented as Java Unicode character literals to Java Unicode character literals (for example: '\u0080'). You can also work around this problem by removing the ".UTF-8" suffix from the locale-specific environment variables; for example, if your machine has a default locale of "en_US.UTF-8", set **LANG** to "en_US".

AMI and xcin problems when exporting displays

If you are using AMI and xcin in a cross-platform environment, there might be a problem if you try to export the display between a 32-bit and a 64-bit system, or between a big-endian and a little-endian system. If you have this problem, upgrade to the latest version of AMI and xcin.

Arabic characters and Matrox video cards

Intel 32-bit platforms only

For Arabic text users, when using Linux with a Matrox video card and acceleration enabled, distortion of characters can be seen when using drawString to display large fonts. This problem is caused by the driver for those cards. The suggested workaround is to disable acceleration for the device.

Java Desktop API

The Java Desktop API might not work because one or more GNOME libraries are not available.

NullPointerException with the GTK Look and Feel

DBCS environments only

If your application fails with a NullPointerException using the GTK Look and Feel, unset the **GNOME_DESKTOP_SESSION_ID** environment variable.

Unicode Shift_JIS code page alias

Japanese users only

The Unicode code page alias "\u30b7\u30d5\u30c8\u7b26\u53f7\u5316\u8868\ u73fe" for Shift_JIS has been removed. If you use this code page in your applications, replace it with Shift_JIS.

-Xshareclasses:<options>

Shared classes cache and control files are not compatible with previous releases, for example between WebSphere Real Time for RT Linux 2.0 SR 3 and previous releases.

Java Kernel installation

The Java kernel aims to reduce the startup time imposed by an application when it finds that the installed JRE needs an update. When this situation occurs, the Java kernel automatically downloads only the Java components that are needed directly from the Sun Web site. This automated download is currently not possible with the IBM implementation of this Sun Java update.

Java Deployment Toolkit

The toolkit implements the JavaScript™ DeployJava.js, which can be used to automatically generate any HTML needed to deploy applets and Java Web Start applications. However, the automatic generation is not possible with the IBM release of Java, because the process involves downloading and running the specific JRE from a public site, using public functions.

Supported browsers for Java plug-ins

Next-generation Java plug-ins are supported only on the Firefox 3 and Firefox 3.5 browsers on Linux.

Supported platforms for Java plug-ins

Support for Java plug-ins and Java Web Start on Linux AMD64 systems is not available at this time.

Expired GTE Cybertrust Certificate

The IBM Runtime Environment for Java contains an expired GTE CyberTrust Certificate in the `CACERTS` file for compatibility reasons. The `CACERTS` file is provided as a default truststore. Some common public certificates are provided as a convenience.

If no applications require the certificate, it can be left in the `CACERTS` file. Alternatively, the certificate can be deleted. If applications do require the certificate, modify them to use the newer GTE CyberTrust Global root certificate that expires in 2018.

This certificate might be removed for later versions of IBM WebSphere Real Time for RT Linux.

Appendix B. RMI-IIOP Programmer's Guide

Discusses how to write Java Remote Method Invocation (RMI) programs that can access remote objects by using the Internet Inter-ORB Protocol (IIOP).

Copyright information

This edition of the RMI-IIOP user guide applies to the RMI-IIOP, and to all subsequent releases, modifications, and Service Refreshes, until otherwise indicated in new editions.

What are RMI, IIOP, and RMI-IIOP?

The basic concepts behind RMI-IIOP and other similar technologies.

RMI

With RMI, you can write distributed programs in the Java programming language. RMI is easy to use, you do not need to learn a separate interface definition language (IDL), and you get Java's inherent "write once, run anywhere" benefit. Clients, remote interfaces, and servers are written entirely in Java. RMI uses the Java Remote Method Protocol (JRMP) for remote Java object communication. For a quick introduction to writing RMI programs, see the RMI tutorial Web page: http://java.sun.com/docs/books/tutorial/rmi, which describes writing a simple "Hello World" RMI program.

RMI lacks interoperability with other languages, and, because it uses a non-standard communication protocol, cannot communicate with CORBA objects.

IIOP, CORBA, and Java IDL

IIOP is CORBA's communication protocol. It defines the way bits are sent over a wire between CORBA clients and servers. CORBA is a standard distributed object architecture developed by the Object Management Group (OMG). Interfaces to remote objects are described in a platform-neutral interface definition language (IDL). Mappings from IDL to specific programming languages are implemented, binding the language to CORBA/IIOP.

The Java Standard Edition v6 CORBA/IIOP implementation is known as Java IDL. Along with the IDL to Java (idlj) compiler, Java IDL can be used to define, implement, and access CORBA objects from the Java programming language.

The Java IDL Web page: http://java.sun.com/j2se/1.5.0/docs/guide/idl/index.html, gives you a good, Java-centric view of CORBA/IIOP programming. To get a quick introduction to writing Java IDL programs, see the Getting Started: Hello World Web page: http://java.sun.com/j2se/1.5.0/docs/guide/idl/GShome.html.

RMI-IIOP

Previously, Java programmers had to choose between RMI and CORBA/IIOP (Java IDL) for distributed programming solutions. Now, by adhering to a few restrictions (see “Restrictions when running RMI programs over IIOP” on page 448), RMI server objects can use the IIOP protocol, and communicate with CORBA client objects written in any language. This solution is known as RMI-IIOP. RMI-IIOP combines RMI ease of use with CORBA cross-language interoperability.

Using RMI-IIOP

This section describes how to use the IBM RMI-IIOP implementation.

The rmic compiler

Reference information about the rmic compiler.

Purpose

The rmic compiler generates IIOP stubs and ties, and emits IDL, in accordance with the Java Language to OMG IDL Language Mapping Specification: http://www.omg.org/cgi-bin/doc?formal/01-06-07.

Parameters

-iiop

Generates stub and tie classes. A stub class is a local proxy for a remote object. Clients use stub classes to send calls to a server. Each remote interface requires a stub class, which implements that remote interface. The remote object reference used by a client is a reference to a stub. Tie classes are used on the server side to process incoming calls, and dispatch the calls to the correct implementation class. Each implementation class requires a tie class.

Stub classes are also generated for abstract interfaces. An abstract interface is an interface that does not extend java.rmi.Remote, but has methods that throw either java.rmi.RemoteException or a superclass of java.rmi.RemoteException. Interfaces that do not extend java.rmi.Remote and have no methods are also abstract interfaces.

-poa

Changes the inheritance from org.omg.CORBA_2_3.portable.ObjectImpl to org.omg.PortableServer.Servant. This type of mapping is nonstandard and is not specified by the Java Language to OMG IDL Mapping Specification: http://www.omg.org/cgi-bin/doc?formal/01-06-07.

The PortableServer module for the Portable Object Adapter (POA) defines the native Servant type. In the Java programming language, the Servant type is mapped to the Java org.omg.PortableServer.Servant class. The class serves as the base class for all POA servant implementations. It provides a number of methods that can be called by the application programmer, as well as methods that are called by the POA itself and might be overridden by the user to control aspects of servant behavior.

Valid only when the **-iiop** option is present.

-idl

Generates OMG IDL for the classes specified and any classes referenced. This option is required only if you have a CORBA client written in another language that needs to talk to a Java RMI-IIOP server.

Tip: After the OMG IDL is generated using **rmic -idl**, use the generated IDL with an IDL-to-C++ or other language compiler, but not with the IDL-to-Java language compiler. "Round tripping" is not recommended and should not be necessary. The IDL generation facility is intended to be used with other languages. Java clients or servers can use the original RMI-IIOP types.

IDL provides a purely declarative means of specifying the API for an object. IDL is independent of the programming language used. The IDL is used as a specification for methods and data that can be written in and called from any language that provides CORBA bindings. Java and C++ are such languages. For a complete description, see the Java Language to OMG IDL Mapping Specification: http://www.omg.org/cgi-bin/doc?formal/01-06-07.

Restriction: The generated IDL can be compiled using only an IDL compiler that supports the CORBA 2.3 extensions to IDL.

-always
Forces regeneration even when existing stubs, ties, or IDL are newer than the input class. Valid only when **-iiop** or **-idl** options are present.

-noValueMethods

Ensures that methods and initializers are not included in valuetypes emitted during IDL Generation. Methods and initializers are optional for valuetypes and are otherwise omitted.

Only valid when used with **-idl** option.

-idlModule *<fromJavaPackage[.class]> <toIDLModule>*

Specifies IDLEntity package mapping. For example: `-idlModule sample.bar my::real::idlmod`.

Only valid when used with **-idl** option.

-idlFile *<fromJavaPackage[.class]> <toIDLModule>*

Specifies IDLEntity file mapping. For example: `-idlFile test.pkg.X TEST16.idl`.

Only valid when used with **-idl** option.

More Information

For more detailed information about the rmic compiler, see the RMIC tool page:

- Solaris, Linux, AIX, and z/OS version: http://java.sun.com/javase/6/docs/technotes/tools/solaris/rmic.html
- Windows version: http://java.sun.com/javase/6/docs/technotes/tools/windows/rmic.html

The idlj compiler

Reference information on the idlj compiler.

Purpose

The idlj compiler generates Java bindings from an IDL file. This compiler supports the CORBA Objects By Value feature, which is required for interoperation with RMI-IIOP. It is written in Java, and so can run on any platform.

More Information

For usage information on the idlj compiler, see IDL-to-Java Compiler User's Guide.

Making RMI programs use IIOP

A general guide to converting an RMI application to use RMI-IIOP.

Before you begin

To use these instructions, your application must already use RMI.

Procedure

1. If you are using the RMI registry for naming services, you must switch to CosNaming:
   a. In both your client and server code, create an InitialContext for JNDI. For a Java application use the following code:

      ```
      import javax.naming.*;
      ...
      Context ic = new InitialContext();
      ```

      For an applet, use this alternative code:

      ```
      import java.util.*;
      import javax.naming.*;
      ...
      Hashtable env = new Hashtable();
      env.put("java.naming.applet", this);
      Context ic = new InitialContext(env);
      ```

   b. Modify all uses of RMI registry lookup(), bind(), and rebind() to use JNDI lookup(), bind(), and rebind() instead. Instead of:

      ```
      import java.rmi.*;
      ...
      Naming.rebind("MyObject", myObj);
      ```

      use:

      ```
      import javax.naming.*;
      ...
      ic.rebind("MyObject", myObj);
      ```

2. If you are not using the RMI registry for naming services, you must have some other way of bootstrapping your initial remote object reference. For example, your server code might be using Java serialization to write an RMI object reference to an ObjectOutputStream and passing this to your client code for deserializing into an RMI stub. When doing this in RMI-IIOP, you must also ensure that object references are connected to an ORB before serialization and after deserialization.
   a. On the server side, use the PortableRemoteObject.toStub() call to obtain a stub, then use writeObject() to serialize this stub to an ObjectOutputStream. If necessary, use Stub.connect() to connect the stub to an ORB before serializing it. For example:

```
org.omg.CORBA.ORB myORB = org.omg.CORBA.ORB.init(new String[0], null);
Wombat myWombat = new WombatImpl();
javax.rmi.CORBA.Stub myStub = (javax.rmi.CORBA.Stub)PortableRemoteObject.toStub(myWombat);
myStub.connect(myORB);
// myWombat is now connected to myORB.  To connect other objects to the
// same ORB, use PortableRemoteObject.connect(nextWombat, myWombat);
FileOutputStream myFile = new FileOutputStream("t.tmp");
ObjectOutputStream myStream = new ObjectOutputStream(myFile);
myStream.writeObject(myStub);
```

b. On the client side, use readObject() to deserialize a remote reference to the object from an ObjectInputStream. Before using the deserialized stub to call remote methods, it must be connected to an ORB. For example:

```
FileInputStream myFile = new FileInputStream("t.tmp");
ObjectInputStream myStream = new ObjectInputStream(myFile);
Wombat myWombat = (Wombat)myStream.readObject();
org.omg.CORBA.ORB myORB = org.omg.CORBA.ORB.init(new String[0], null);
((javax.rmi.CORBA.Stub)myWombat).connect(myORB);
// myWombat is now connected to myORB.  To connect other objects to the
// same ORB, use PortableRemoteObject.connect(nextWombat, myWombat);
```

The JNDI approach is much simpler, so it is preferable to use it whenever possible.

3. Either change your remote implementation classes to inherit from javax.rmi.PortableRemoteObject, or explicitly to export implementation objects after creation by calling PortableRemoteObject.exportObject(). For more discussion on this topic, read "Connecting IIOP stubs to the ORB" on page 448.
4. Change all the places in your code where there is a Java cast of a remote interface to use javax.rmi.PortableRemoteObject.narrow().
5. Do not depend on distributed garbage collection (DGC) or use any of the RMI DGC facilities. Use PortableRemoteObject.unexportObject() to make the ORB release its references to an exported object that is no longer in use.
6. Regenerate the RMI stubs and ties using the rmic command with the **-iiop** option. This will produce stub and tie files with the following names:

   _<*implementationName*>_Tie.class
   _<*interfaceName*>_Stub.class
7. Before starting the server, start the CosNaming server (in its own process) using the `tnameserv` command The CosNaming server uses the default port number of 2809. If you want to use a different port number, use the **-ORBInitialPort** parameter.
8. When starting client and server applications, you must specify some system properties. When running an application, you can specify properties on the command line:

   ```
   java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
        -Djava.naming.provider.url=iiop://<hostname>:2809
        <appl_class>
   ```
9. If the client is an applet, you must specify some properties in the applet tag. For example:

   ```
   java.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
   java.naming.provider.url=iiop://<hostname>:2809
   ```

   This example uses the default name service port number of 2809. If you specify a different port in the previous step, you need to use the same port number in the provider URL here. The <*hostname*> in the provider URL is the host name that was used to start the CosNaming server.

Results

Your application can now communicate with CORBA objects using RMI-IIOP.

Connecting IIOP stubs to the ORB

When your application uses IIOP stubs, as opposed to JRMP stubs, you must properly connect the IIOP stubs with the ORB before starting operations on the IIOP stubs (this is not necessary with JRMP stubs). This section discusses the extra 'connect' step required for the IIOP stub case.

The PortableRemoteObject.exportObject() call only creates a Tie object and caches it for future usage. The created tie does not have a delegate or an ORB associated. This is known as explicit invocation.

The PortableRemoteObject.exportObject() happens automatically when the servant instance is created. The servant instance is created when a PortableRemoteObject constructor is called as a base class. This is known as implicit invocation.

Later, when the application calls PortableRemoteObject.toStub(), the ORB creates the corresponding Stub object and associates it with the cached Tie object. But because the Tie is not connected and does not have a delegate, the newly created Stub also does not have a delegate or ORB.

The delegate is set for the stub only when the application calls Stub.connect(orb). Thus, any operations on the stub made before the ORB connection is made will fail.

The Java Language to OMG IDL Mapping Specification (http://www.omg.org/cgi-bin/doc?formal/01-06-07) says this about the Stub.connect() method:

> "The connect method makes the stub ready for remote communication using the specified ORB object orb. Connection normally happens implicitly when the stub is received or sent as an argument on a remote method call, but it is sometimes useful to do this by making an explicit call (e.g., following deserialization). If the stub is already connected to orb (has a delegate set for orb), then connect takes no action. If the stub is connected to some other ORB, then a RemoteException is thrown. Otherwise, a delegate is created for this stub and the ORB object orb."

For servants that are not POA-activated, Stub.connect(orb) is necessary as a required setup.

Restrictions when running RMI programs over IIOP

A list of limitations when running RMI programs over IIOP.

To make existing RMI programs run over IIOP, observe the following restrictions.

- Make sure all constant definitions in remote interfaces are of primitive types or String and evaluated at compile time.
- Do not use Java names that conflict with IDL mangled names generated by the Java-to-IDL mapping rules. See section 28.3.2 of the Java Language to OMG IDL Mapping Specification for more information: http://www.omg.org/cgi-bin/doc?formal/01-06-07
- Do not inherit the same method name into a remote interface more than once from different base remote interfaces.
- Be careful when using names that are identical other than their case. The use of a type name and a variable of that type with a name that differs from the type name in case only is supported. Most other combinations of names that are identical other than their case are not supported.

- Do not depend on runtime sharing of object references to be preserved exactly when transmitting object references to IIOP. Runtime sharing of other objects is preserved correctly.
- Do not use the following features of RMI, which do not work in RMI-IIOP:
  - RMISocketFactory
  - UnicastRemoteObject
  - Unreferenced
  - The Distributed Garbage Collector (DGC) interfaces

Additional information

Information about thread safety, working with other ORBs, the difference between UnicastRemoteObject and PortableRemoteObject, and known limitations.

Servers must be thread safe

Because remote method invocations on the same remote object might execute concurrently, a remote object implementation must be thread-safe.

Interoperating with other ORBs

RMI-IIOP should interoperate with other ORBs that support the CORBA 2.3 specification. It will not interoperate with older ORBs, because older ORBs cannot handle the IIOP encodings for Objects By Value. This support is needed to send RMI value classes (including strings) over IIOP.

Note: Although ORBs written in different languages should be able to interoperate, the Java ORB has not been fully tested with other vendors' ORBs.

When do I use UnicastRemoteObject vs PortableRemoteObject?

Use UnicastRemoteObject as the superclass for the object implementation in RMI programming. Use PortableRemoteObject in RMI-IIOP programming. If PortableRemoteObject is used, you can switch the transport protocol to either JRMP or IIOP during runtime.

Known limitations

- JNDI 1.1 does not support *java.naming.factory.initial*=com.sun.jndi.cosnaming.CNCtxFactory as an Applet parameter. Instead, it must be explicitly passed as a property to the InitialContext constructor. This capability is supported in JNDI 1.2.
- When running the Naming Service on Unix based platforms, you must use a port number greater than 1024. The default port is 2809, so this should not be a problem.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product,

program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1758
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

JIMMAIL@uk.ibm.com
[Hursley Java Technology Center (JTC) contact]

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided “AS IS”, without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common

law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Appendix C. Security Guide

An overview of the security features in the IBM SDK for Java.

Copyright information

This edition of the user guide applies to the security components included with the IBM SDK for Java.

Note: Before using this information and the product it supports, read the general information under Notices.

This edition of the user guide applies to:

- iKeyman
- Java Authentication and Authorization Service (JAAS) v2.0
- IBM Java Certification Path (CertPath) v1.1 Provider
- IBM Java Cryptography Extension (JCE) Provider
- IBM Java Generic Security Service (JGSS) v1.5 Provider
- IBM Java Secure Socket Extension (JSSE) IBM JSSE2 Provider
- IBM PKCS11 Implementation Provider
- IBM Java JCE FIPS Provider
- IBM Simple Authentication and Security Layer (SASL) Provider v1.6
- Key Certificate Management utilities
- Java XML encryption and signatures

and to all subsequent releases and modifications until otherwise indicated in new editions.

Preface

The security components described in this user guide are shipped with the SDK and are not extensions. They provide a wide range of security services through standard Java APIs (except iKeyman). The security components contain the IBM implementation of various security algorithms and mechanisms. IBM does not provide support for any of the IBM Java security components when used with a non-IBM JVM or with non-IBM security providers when used with the IBM JVM.

The IBM SDK also provides a FIPS 140-2 certified cryptographic module, IBMJCEFIPS, implemented as a JCE provider. Applications can comply with the FIPS 140-2 requirements by using the IBMJCEFIPS module.

The CertPath component provides PKIX-compliant certification path building and validation.

The JGSS component provides a generic API that can be plugged in by different security mechanisms. IBM JGSS uses Kerberos V5 as the default mechanism for authentication and secure communication.

The JAAS component provides a means for principal-based authentication and authorization.

The JCE framework has two providers: IBMJCE is the pre-registered default provider; IBMJCEFIPS is optional.

JSSE is the Java implementation of the SSL and TLS protocols. The JSSE pre-registered default provider is IBMJSSE2.

IBM Java Simple Authentication and Security Layer, or SASL, is an Internet standard (RFC 2222) that specifies a protocol for authentication and optional establishment of a security layer between client and server applications.

The Java security configuration file does not refer to the Sun provider. The IBM JCE provider has replaced the Sun provider. The JCE supplies all the signature handling message digest algorithms that were previously supplied by the Sun provider. It also supplies the IBM secure random number generator, IBMSecureRandom, which is a real Random Number Generator. SHA1PRNG is a Pseudo Random Number Generator and is supplied for code compatibility. SHA1PRNG is not guaranteed to produce the same output as the SUN SHA1PRNG.

In the IBM SDK v1.4.1, the following options were added to the **java.security.debug** property to help you debug Java Cryptography Architecture (JCA)-related problems:

provider
: Displays each provider request and load, provider add, and provider remove. It also displays the related exception when a provider load fails.

algorithm
: Displays each algorithm request, which provider has supplied the algorithm, and the implementing class name.

:stack You can append this option to either of **algorithm** - or **provider**. When you request an algorithm, a stack trace is displayed. Use this stack trace to determine the code that has requested the algorithm. This option also prints the stack trace for exceptions that are caught or converted.

:thread
: Adds the thread id to all debug message lines. You can use this option together with all the other debug options.

An example of a valid option string is "provider, algorithm:stack".

In this guide, there is a 'What's new' section for each component. This information is provided to help you with migration.

General information about IBM security providers

Overview of the security providers tested with the IBM SDK.

The IBM SDK v6 has been tested with the following default security providers:

- security.provider.1=com.ibm.jsse2.IBMJSSEProvider2
- security.provider.2=com.ibm.crypto.provider.IBMJCE
- security.provider.3=com.ibm.security.jgss.IBMJGSSProvider
- security.provider.4=com.ibm.security.cert.IBMCertPath

- security.provider.5=com.ibm.security.sasl.IBMSASL
- security.provider.6=com.ibm.xml.crypto.IBMXMLCryptoProvider
- security.provider.7=com.ibm.xml.enc.IBMXMLEncProvider
- security.provider.8=org.apache.harmony.security.provider.PolicyProvider
- security.provider.9=com.ibm.security.jgss.mech.spnego.IBMSPNEGO

You can add other IBM security providers either statically or from inside your Java application's code. To add a new provider statically, edit a Java security properties file (for example, `java.security`). To add a new provider from your application's code, use the methods of the java.security.Security class (for example, `java.security.Security.addProvider()`).

You can also add this IBM security provider, com.ibm.crypto.fips.provider.IBMJCEFIPS.

iKeyman tool

The iKeyman utility is a tool for key databases containing digital certificates and keys.

With iKeyman, you can:

- Create and manage key databases.
- Create self-signed digital certificates for testing.
- Add certificate authority (CA) and intermediate certificates.
- Transfer certificates between key databases.
- Create certificate requests and receive a digital certificate issued by a CA in response to a request.
- Create, import and export symmetric keys.

To load a Java cryptographic token on SLES10 using iKeyman, you must rename the cryptography library from `/usr/lib/opencryptoki/libopencryptoki.so.0.0.0:0` to `/usr/lib/opencryptoki/libopencryptoki.so`. If `/usr/lib/opencryptoki/libopencryptoki.so` is already linked, you must delete it before the renaming.

History of changes

Version 6

Improved error messages and command help.

PKCS#11 access provided by IBMPKCS11Impl Provider rather than IBMPKCS11 Provider and a native library.

Support for symmetric key management (only supported by JCEKS and PKCS#11 key databases).

Use of the Java CMS Provider for CMS key databases rather than a native library.

Documentation

For more information, including information about the iKeyman GUI, see the *iKeyman User Guide* at: http://www.ibm.com/developerworks/java/jdk/security/index.html.

Java Authentication and Authorization Service (JAAS) V2.0

The Sun Microsystems Java platform provides a means to enforce access controls based on *where* code came from and *who signed* it. These access controls are needed because of the distributed nature of the Java platform where, for example, a remote applet can be downloaded over a public network and then run locally.

However, before SDK v1.4.0, the Java platform did not provide a way to enforce similar access controls based on *who runs* the code. To provide this type of access control, the Java security architecture requires the following:

- Additional support for authentication (determining who is running the code)
- Extensions to the existing authorization components to enforce new access controls based on who was authenticated

The Java Authentication and Authorization Service (JAAS) framework provides these enhancements.

For a general overview of JAAS, see the Sun Web site: http://java.sun.com/products/jaas.

Differences between IBM and Sun versions of JAAS

IBM implementations are contained in the com.ibm.* package instead of the com.sun.* package.

Further reading

For detailed information, including API documentation and samples, see the developerWorks Web site at http://www.ibm.com/developerworks/java/jdk/security/index.html. This site contains the *LoginModule Developer's Guide* and sample code in "HelloWorld.tar".

History of changes

A history of the changes to the Java Authentication and Authorization Service (JAAS) since it was added to the SDK.

Changes for Version 6

Added a new JAAS login module which enables users to perform authentication using credentials stored in an LDAP directory service. The new module provides LDAP support for existing JAAS-enabled applications.

Original release

New to the IBM 32-bit SDK for Linux on Intel architecture

The original release of JAAS for Linux and the Java Platform included the following login module and principal classes:

- com.ibm.security.auth.module.LinuxLoginModule
- com.ibm.security.auth.LinuxPrincipal
- com.ibm.security.auth.LinuxNumericGroupPrincipal
- com.ibm.security.auth.LinuxNumericUserPrincipal

These original platform-dependent principal classes will be replaced by a set of platform-independent principal classes in future releases of JAAS for Linux. To ease migration, this version of JAAS contains the original set as well as the new set of principal classes. Also included is a new login module called com.ibm.security.auth.module.LinuxLoginModule2000, which has the same function as LinuxLoginModule but references the new set of principals. Additional principal classes have been included to facilitate the writing of new login modules.

You are encouraged to use the new set of principals when developing applications that use JAAS. Previously developed applications will be compatible with this version as well as future versions of JAAS released for the SDK v1.4.0.

If migrating applications to the new set of principals is desired, then most changes encountered will be in JAAS policy and configuration files rather than in the applications. Refer to the following table for guidance.

Table 27. New class names

Original class	Replaced by
LinuxPrincipal	UsernamePrincipal
LinuxNumericGroupPrincipal	GroupIDPrincipal PrimaryGroupIDPrincipal
LinuxNumericUserPrincipal	UserIDPrincipal
n/a	DomainPrincipal
n/a	DomainIDPrincipal
n/a	ServerPrincipal
n/a	WkstationPrincipal
LinuxLoginModule	LinuxLoginModule2000

Principal classes are found in the com.ibm.security.auth package while the login module is found in the com.ibm.security.auth.module package. Check the JAAS API documentation (Javadoc information) for more information on the new principal classes.

For example, this JAAS policy grant block:

```
grant Principal com.ibm.security.auth.LinuxPrincipal "bob",
      Principal com.ibm.security.auth.LinuxNumericUserPrincipal
           "727",
      Principal com.ibm.security.auth.LinuxNumericGroupPrincipal
           "12" {
   permission java.util.PropertyPermission "java.home", "read";
};
```

is replaced by:

```
grant Principal com.ibm.security.auth.UsernamePrincipal "bob",
      Principal com.ibm.security.auth.UserIDPrincipal "727",
      Principal com.ibm.security.auth.GroupIDPrincipal "12" {
   permission java.util.PropertyPermission "java.home", "read";
};
```

Java Certification Path (CertPath)

The Java Certification Path API provides interfaces and abstract classes for creating, building, and validating certification paths (also known as "certificate chains").

Differences between IBM and Sun versions of CertPath

The IBM CertPath classes differ from the Sun version in the following ways:

- The IBM CertPath provider is in the package com.ibm.security.cert.
- The IBM CertPath provider is called "IBMCertPath". Sun does not have a separate provider for CertPath; CertPath is already supported by the "SUN" provider.
- To enable CRL Distribution Points extension checking, use the system property **com.ibm.security.enableCRLDP**. The system property used by the Sun version is **com.sun.security.enableCRLDP**.
- When checking the CRL Distribution Points extension of the certificate, the Sun CertPath provided retrieves the CRL only if the CRL location is specified as an HTTP URL value inside the extension. The IBM provider recognizes both HTTP and LDAP URLs.
- The IBM implementation of CertPath supports the processing of both complete CRLs and delta CRLs. Setting the com.ibm.security.enableDELTACRL system property to true enables the use of both delta CRLs and complete CRLs if revocation checking is enabled by the caller. If com.ibm.security.enableDELTACRL is set to false, or is not set, only complete CRLs are used.

Documentation

For detailed information, including API documentation and samples, see the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

History of changes

A history of the changes to CertPath since it was added to the SDK.

Changes for Version 6

- Enhanced CRL validation and CRL processing to more closely comply with the PKIX Certificate and CRL Profile (RFC 3280) Section 6.3, entitled “CRL Validation”. This processing is performed only if revocation checking is enabled by the caller inside the supplied PKIXParameters object.
- Added the com.ibm.security.enableDELTACRL system property to use both delta CRLs and complete CRLs if revocation checking is enabled by the caller. If com.ibm.security.enableDELTACRL is set to false, or is not set, only complete CRLs are used.
- Added the com.ibm.security.enableAIAEXT system property to use LDAP URIs found in any Authority Information Access extensions in certificates on the certificate path. For each LDAP URI found, an LDAPCertStore object is created and added to the collection of CertStores used to locate other certificates needed to build the certificate path.

Changes for Version 5.0

- Added support for checking a certificate's revocation status based on On-Line Certificate Status Protocol (OCSP).
- Added a new constructor and a public API in the TrustAnchor class:

```
public TrustAnchor(X500Principal caPrincipal, PublicKey pubKey, byte[] nameConstraints);
public final X500Principal getCA();
```

- Added new public APIs in X509CertSelector:

```
public X500Principal getIssuer();
public void setIssuer(X500Principal issuer);
public X500Principal getSubject();
public void setSubject(X500Principal subject);
```

- Added new public APIs in X509CRLSelector:

```
public void setIssuers(Collection issuers);
public void addIssuer(X500Principal issuer);
public Collection getIssuers();
```

- Changed the PolicyQualifier class to non-final. The public APIs have changed to be final.

Changes for Version 1.4.2

- Improved the performance of the IBM CertPath provider.
- Added limited support for the CRL Distribution Points extension.
- Added caching to cache lookups in the IBM LDAP CertStore.

Changes for Version 1.4.1, Service Refresh 1

- The trusted certificate that acts as TrustAnchor can be an X.509 v1 certificate.
- When you specify the certificate's subject or issuer name as a String in X509CertSelector, the search for a matched certificate mechanism checks only the name value and ignores the tag type.

Changes for Version 1.4.0

- Certificates from CertificatePair entry can be retrieved from LDAP type certstore.
- Changed the framework package name from javax.security.cert to java.security.cert. The old framework package is still supported.

Java Cryptography Extension (JCE)

The Java Cryptography Extension (JCE) provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects. JCE supplements the Java platform, which already includes interfaces and implementations of message digests and digital signatures.

You can obtain unrestricted jurisdiction policy files from http://www.ibm.com/developerworks/java/jdk/security/index.html. The policy files are replaced with restricted policy files when you upgrade your SDK. Before upgrading your SDK, make a backup of your policy files. After upgrading your SDK, install your backup policy files if you need an unrestricted policy.

The v1.4.2 unrestricted (and restricted) jurisdiction policy files are suitable for use with v5.0 and later. The v1.4.1 files are not suitable.

Differences between IBM and Sun versions of JCE

The com.sun.* packages are reimplemented by IBM and renamed com.ibm.* packages.

The IBM version of JCE differs from the Sun version in the following ways:

- The com.sun.crypto.* packages are reimplemented by IBM and renamed com.ibm.crypto.* packages.

- The IBM JCE provider replaces the Sun providers sun.security.provider.Sun, com.sun.rsajca.Provider, and com.sun.crypto.provider.SunJCE.
- IBM provides more algorithms than Sun does:

  Cipher algorithms

  AES

  Blowfish

  DES

  El Gamal

  Mars

  ARCFOUR

  PBE with MD2 and DES

  PBE with MD2 and Triple DES

  PBE with MD2 and RC2

  PBE with MD5 and DES

  PBE with MD5 and Triple DES

  PBE with MD5 and RC2

  PBE with SHA1 and DES

  PBE with SHA1 and TripleDES

  PBE with SHA1 and RC2

  PBE with SHA1 and 40-bit RC2

  PBE with SHA1 and 128-bit RC2

  PBE with SHA1 and 40-bit RC4

  PBE with SHA1 and 128-bit RC4

  PBE with SHA1 and 2-key Triple DES

  PBE with SHA1 and 3-key Triple DES

  RC2

  RC4

  RSA encryption/decryption

  RSA encryption/decryption with OAEP Padding

  Seal

  Triple DES

  Signature algorithms

  SHA1 with RSA

  SHA2 with RSA

  SHA3 with RSA

  SHA5 with RSA

  MD5 with RSA

  MD2 with RSA signature

  SHA1 with DSA signature

Message digest algorithms

SHA1

SHA2

SHA3

SHA5

MD5

MD2

Message authentication code (MAC)

Hmac/SHA1

Hmac/MD5

Hmac/SHA2

Hmac/SHA3

Hmac/SHA5

Key agreement algorithm

DiffieHellman

Random number generation algorithms

IBMSecureRandom

IBM SHA1PRNG

Key Store

JCEKS

JKS

PKCS12KS

Documentation

For detailed information, including API documentation and samples, see the developerWorks Web site at http://www.ibm.com/developerworks/java/jdk/security/index.html.

History of changes

A history of the changes to the Java Cryptography Extension (JCE) since it was added to the SDK.

Changes for Version 6

- Added CipherTextStealing mode.
- Added ISO10126Padding support.
- Added PBKDF2HmacSHA1Factory and PBKDF2KeyImpl.
- Added supporting classes for Tls use with JSSE.

Changes for Version 5.0, Service Refresh 4

- Added the EL Gamal Cipher and supporting classes.

Changes for Version 5.0

- Added RSA with OAEP Padding.
- Added the SHA2withRSA, SHA3withRSA and SHA5withRSA signature algorithms.

- Added the HmacSHA2, HmacSHA3, HmacSH5 MAC algorithms.
- Added the ARCFOUR encryption algorithm.

Changes for Version 1.4.2

- Added the SHA2, SHA3 and SHA5 hashing algorithms.
- Added the SHA1PRNG algorithm for generating pseudo random numbers.

Changes for Version 1.4.0

- Added the AES cipher algorithm.
- Changed to strong cryptography by default. Unlimited cryptography is available.
- Provider authentication of the JCE framework no longer required.
- JCE is now shipped with the Java SDK v1.4 on all platforms.

Java Generic Security Service (JGSS)

Java Generic Security Service (JGSS) API provides secure exchange of messages between communicating applications.

JGSS is an API framework that uses Kerberos V5 as the underlying default security mechanism. The API is a standardized abstract interface under which you can plug different security mechanisms that are based on private-key, public-key, and other security technologies.

JGSS shields secure applications from the complexities and peculiarities of the different underlying security mechanisms. JGSS provides identity and message origin authentication, message integrity, and message confidentiality. JGSS also features an optional Java Authentication and Authorization Service (JAAS) Kerberos login interface, and authorization checks. JAAS augments the access control features of Java, which is based on CodeSource with access controls based on authenticated principal identities.

Differences between the IBM and Sun versions of JGSS

The IBM version of JGSS differs from the Sun version in the following ways:

- The com.sun.* packages are reimplemented by IBM and renamed com.ibm.* packages.
- The format of the parameters passed to the Java tools kinit, ktab, and klist is different from the equivalent tools provided in the Sun version of JGSS.

Documentation

For detailed information about JGSS, including API documentation and samples, see the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

History of changes

A history of the changes to the Java Generic Security Service (JGSS) since it was added to the SDK.

Change for Version 6

Added CipherTextStealing mode.

Changes for Version 5.0, Service Refresh 1

Added AES as a supported algorithm type

These additional algorthims can be set in the `krb5.conf` file under [libdefault] as follows:

```
default_tkt_enctypes = aes128-cts-hmac-sha1-96
default_tkt_enctypes = aes256-cts-hmac-sha1-96
default_tgs_enctypes = aes128-cts-hmac-sha1-96
default_tgs_enctypes = aes256-cts-hmac-sha1-96

default_checksum   = hmac-sha1-96-aes128
default_checksum   = hmac-sha1-96-aes256
```

Changes for Version 5.0

TCP or UDP Preference Configuration

Added JSE support for the **udp_preference_limit** property in the Kerberos configuration file (krb5.ini). When sending a message to the KDC, the JSE Kerberos library will use TCP if the size of the message is above **udp_preference_list**. If the message is smaller than **udp_preference_list**, UDP will be tried up to three times. If the KDC indicates that the request is too big, the JSE Kerberos library will use TCP.

IPv6 support in Kerberos

Added JSE support for IPv6 addresses in Kerberos tickets. Before v5.0, only IPv4 addresses were supported in tickets.

TGT Renewals

Added support for Ticket Granting Ticket (TGT) renewal to the Java Authentication and Authorization Server (JAAS) Kerberos login module, Krb5LoginModule. This support allows long-running services to renew their TGTs automatically without user interaction or requiring the services to restart. With this feature, if Krb5LoginModule obtains an expired ticket from the ticket cache, the TGT will be automatically renewed and be added to the Subject of the caller who requested the ticket. If the ticket cannot be renewed for any reason, Krb5LoginModule will use its configured callback handler to retrieve a username and password to acquire a new TGT.

To use this feature, configure Krb5LoginModule to use the ticket cache and set the newly introduced **renewTGT** option to true. Here is an example of a JAAS login configuration file that requests TGT renewal:

```
server {
  com.ibm.security.auth.module.Krb5LoginModule required
        principal=principal@your_realm
  useDefaultCcache=TRUE
   renewTGT=true;
};
```

Note that if **renewTGT** is set to true, **useDefaultCcache** must also be set to true; otherwise, it results in a configuration error.

Changes for Version 1.4.2

Configurable Kerberos Settings

You can provide the name and realm settings for the Kerberos Key Distribution Center (KDC) either from the Kerberos configuration file or by using the system properties files java.security.krb5.kdc and java.security.krb5.realm. You can also specify the boolean option **refreshKrb5Config** in the entry for Krb5LoginModule in the JAAS

configuration file. If you set this option to `true`, the configuration values will be refreshed before the login method of the Krb5LoginModule is called.

Added support for Slave Kerberos Key Distribution Center
Kerberos uses slave KDCs so that, if the master KDC is unavailable, the slave KDCs will respond to your requests. In previous releases, Kerberos tried the master KDC only and gave up if there was no response in the default KDC timeout period.

Added support for TCP for Kerberos Key Distribution Center Transport
Kerberos uses UDP transport for ticket requests. In cases where Kerberos tickets exceed the UDP packet size limit, Kerberos supports automatic fallback to TCP. If a Kerberos ticket request using UDP fails and the KDC returns the error code `KRB_ERR_RESPONSE_TOO_BIG`, TCP becomes the transport protocol.

Kerberos Service Ticket in the Subject's Private Credentials
The Kerberos service ticket is stored in the Subject's private credentials. This gives you access to the service ticket so that you can use it outside the JGSS (for example, in native applications or for proprietary uses). In addition, you can reuse the service ticket if the application tries to establish a security context to the same service again. The service ticket should be valid for it to be reusable.

Changes for Version 1.4.1

- Added wrappers for the klist, kinit, and ktab Java tools. These wrappers call the relevant tool classes so that you do not have to remember the full package name.

IBMJSSE2 Provider

The Java Secure Socket Extension (JSSE) is a Java package that enables secure internet communications. The package implements a Java version of SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols. It includes functions for data encryption, server authentication, message integrity, and optional client authentication.

By abstracting the complex underlying security algorithms and "handshaking" mechanisms, JSSE minimizes the risk of creating subtle but dangerous security vulnerabilities. Also, it simplifies application development by serving as a building block that you can integrate directly into your applications. Using JSSE, you can provide for the secure passage of data between a client and a server running any application protocol (such as HTTP, Telnet, NNTP, and FTP) over TCP/IP.

The FIPS provider included with the SDK is undergoing certification with the US Government. The certification progress is available on the CSRC Web site: http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140InProcess.pdf.

Documentation

For detailed information, including API documentation and samples, see http://www.ibm.com/developerworks/java/jdk/security/index.html.

Differences between the IBMJSSE Provider and the IBMJSSE2 Provider

The IBMJSSE2 Provider, which was introduced in the v1.4.2 SDK, has replaced the IBMJSSE Provider. Although they are nearly equivalent, there are differences between the two providers.

The now-discontinued IBMJSSE Provider and the IBMJSSE2 Provider differ in the following ways:

- The IBMJSSE2 Provider is called com.ibm.jsse2.IBMJSSEProvider2.
- The HTTPS protocol handler for the IBMJSSE2 Provider is called com.ibm.net.ssl.www2.protocol.Handler. The com.ibm.net.ssl.internal.www.protocol.Handler and the com.ibm.net.ssl.www.protocol.Handler protocol handlers have been removed.
- The IBMJSSE2 Provider does not support the com.ibm.net.ssl framework. Use the javax.net.ssl framework instead.
- The IBMJSSE2 Provider does not support the SSL version 2 protocol. However, the server side of a JSSE2 connection does accept the SSLv2Hello protocol.
- The AES_256 ciphers require the installation of the JCE Unlimited Strength Jurisdiction Policy. The old IBMJSSE Provider did not use JCE for its cryptographic support and therefore did not require these files.
- The IBMJSSE2 Provider requires a JCE Provider for its cryptography.
- The IBMJSSE2 Provider does not build the server's private key certificate chain from the trusted keystore. The trusted certificates must be added to the server's private key to complete the chain. This is an incompatible change.
- The IBMJSSE2 Provider considers a certificate trusted if you have the private key.
- The HTTPS protocol handler for the IBMJSSE2 Provider performs hostname verification and rejects requests where the host to connect to and the server name from the certificate do not match. A HostnameVerification implementation called com.ibm.jsse2.HostnameVerifierIgnore is provided. com.ibm.jsse2.HostnameVerifierIgnore always accepts the connection even when a mismatch occurs.
- Tracing no longer requires a separate debug jar.
- The class com.ibm.jsse.SSLContext, which in IBMJSSE is used to access secure tokens, has been removed. Use the hardware crypto support in IBMJSSE2 instead. See the documentation on the developerWorks Web site http://www.ibm.com/developerworks/java/jdk/security/index.html for details.
- The IBMJSSEFIPS Provider has been removed. JSSE FIPS support is supported in the IBMJSSE2 Provider and no separate jar is required. See the documentation on the developerWorks Web site http://www.ibm.com/developerworks/java/jdk/security/index.html for instructions how to set up JSSE to run in FIPS mode.

Differences between the IBMJSSE2 Provider and the Sun version of JSSE

Although they are nearly equivalent, there are differences between the IBMJSSE2 Provider and the Sun JSSE Provider.

The IBMJSSE2 Provider differs from the Sun JSSE in the following ways:

- The IBM JSSE Provider is called com.ibm.jsse2.IBMJSSEProvider2.
- The IBM KeyManagerFactory is called IbmX509 or NewIbmX509.

- The IBM TrustManagerFactory is called IbmX509 or IbmPKIX.
- The IBM HTTPS protocol handler is called com.ibm.net.ssl.www2.protocol.Handler.
- IBMJSSE2 does not support the com.sun.net.ssl framework; use the javax.net.ssl framework instead.
- You can use PKIK revocation checking by setting the system property *com.ibm.jsse2.checkRevocation* to "true".
- The IBM implementation supports the following protocols for the engine class SSLContext, for the API setEnabledProtocols in the SSLSocket, and for SSLServerSocket classes:
  - SSL
  - SSLv3
  - TLS
  - TLSv1
  - SSL_TLS

  The IBM implementation *does not* support the "SSLv2Hello"protocol. The IBM implementation supports the SSL v2 protocol. You can use the IBM SSLContext `getInstance()` factory method to control which protocols are enabled for an SSL connection. Using the SSLContext `getInstance()` or the `setEnabledProtocols()` methods provides the same result. With the Sun version of JSSE, the protocol is controlled through `setEnabledProtocols()`.
- IBM and Sun support different cipher suites.
- The IBM JSSE TrustManager does not allow anonymous ciphers. To handshake with an anonymous cipher, a custom TrustManager that allows anonymous ciphers must be provided.
- When a null KeyManager is passed to SSLContext, the IBM JSSE KeyManagerFactory implementation first checks the system properties. Next, if it exists, jssecacerts is checked. Finally, the `cacerts` file is used to find the key material. The Sun version of JSSE creates an empty KeyManager.
- The IBM JSSE X509TrustManager and X509KeyManager throws an exception if any one of three conditions are true:
  - The TrustStore or KeyStore specified by the system properties does not exist.
  - The password is incorrect.
  - The keystore type is inappropriate for the actual keystore.

  The Sun version of X509TrustManager creates a default TrustManager or KeyManager with an empty keystore.
- The IBM JSSE implementation verifies the entire server or client certificate chain, including trusted certificates. For example, if a trusted certificate has expired, the handshake fails, even though the expired certificate is trusted. The Sun version of JSSE verifies the certificate chain up to the trusted certificate. Verification stops when it reaches a trusted certificate and the trusted certificate and beyond are not verified.
- The IBM JSSE implementation returns the same set of supported ciphers for the methods `getDefaultCiphersSuites()` and `getSupportedCipherSuites()`. The Sun version of `getDefaultCipherSuites()` returns the list of cipher suites that provide confidentiality protection and server authentication (that is, no anonymous cipher suites). Sun's `getEnabledCipherSuites()` returns the entire list of cipher suites that Sun supports.
- For the Sun implementation, DSA server certificates can use only *_DH*_* cipher suites. For the IBM implementation, if the server has a DSA certificate only, and

only RSA* ciphers are enabled, the connection succeeds with an RSA cipher. DSA will be used for authentication and ephemeral RSA will be used for the key exchange.

- To use a hardware keystore or truststore on an IBM hardware crypto provider with system properties, set the **javax.net.ssl.keyStoreType** and **javax.net.ssl.trustStoreType** system properties to "PKCS11IMPLKS". For the Sun implementation, the system property is set to "PKCS11".
- The IBM and Sun JSSE Providers can be enabled to run in FIPS mode although they are enabled differently.

History of changes

A history of the changes to the IBMJSSE2 Provider since it was added to the SDK.

Changes for Version 6

- Added default SSLContext methods to SSLContext.
- Added the new SSLParameters class.
- Removed pluggability restrictions from the JSSE Framework.

Changes for Version 5.0, Service Refresh 5

- When IBMJSSE2 is used as a server, if the SSLv3 protocol is to be used for the handshake, it will no longer agree to use any of the AES cipher suites. Previously, the selection of the cipher suite was independent of the protocol selected so you could do an old-style SSLv3 handshake with a more modern AES cipher suite. The TLS protocol is not affected by this change. This change was required to support Microsoft Vista clients.

Changes for Version 5.0

- Removed the IBMJSSE Provider. Use the IBMJSSE2 Provider instead.
- The IBMJSSE2 implementation supports only IBM hardware crypto providers. Applications that ran in v1.4.2 using the IBMPKCS11Impl provider will now be required to use a configuration file in order to run successfully.
- SSLEngine (non-blocking I/O) allows SSL/TLS applications to choose their own I/O and compute models.
- Enhanced TrustManager support for HTTP/HTTPS.
- New and updated Methods and Classes.
- Kerberos cipher suites are available, if supported by the operating system.

Changes for Version 1.4.2.

The IBMJSSE2 Provider is new for Version 1.4.2.

IBMPKCS11Impl Provider

The IBMPKCS11Impl Provider uses the Java Cryptography Extension (JCE) and Java Cryptography Architecture (JCA) frameworks to add the ability to use hardware cryptography through the Public Key Cryptographic Standards #11 (PKCS #11) standard.

This provider takes advantage of hardware cryptography in the existing JCE architecture. It gives Java programmers the significant security and performance advantages of hardware cryptography with minimal changes to existing Java

applications. Because the complexities of hardware cryptography are handled in the normal JCE, advanced security and performance using hardware cryptographic devices is available readily.

PKCS#11 is a standard that provides a common application interface to cryptographic services on various platforms through several hardware cryptographic devices. See the IBMPKCS11Impl provider user guide for a list of supported devices.

Remember: Use the correct PKCS11 configurations files for Version 6 of the SDK. You can download sample configuration files from http://www.ibm.com/developerworks/java/jdk/security/60/.

Differences between IBM and Sun versions of IBMPKCS11Impl

- The Sun keystore is named PKCS11 and the IBM keystore is called IBMPKCS11KS.
- The Sun keystore requires that all trusted certificates have the attribute **CKA_TRUSTED** set to true. The IBM keystore assumes that any certificates on the device are trusted. The IBM keystore can work with data that was saved using the Sun keystore. However, the Sun keystore might not be able to work with data saved using the IBM keystore.

Documentation

For detailed information, including API documentation, see the developerWorks Web site at http://www.ibm.com/developerworks/java/jdk/security/index.html.

History of changes

A history of the changes to the IBMPKCS11Impl Provider since it was added to the SDK.

Changes for Version 6

- Added support for TLS/SSL key generations, ECDH key agreement, and ECDSA signature algorithms including SHA1withECDSA, SHA256withECDSA, SHA384withECDSA, and SHA512withECDSA.
- Added session management so that applications can concurrently execute multi-part crypto operations within a single provider.

Changes for Version 5.0

Updated IBMPKCS11Impl to allow more algorithms and to allow the Sun 5.0 methods of initialization of the provider. The new algorithms are:

- AES
- Diffie-Hellman
- RC4, also known as ArcFour
- Blowfish
- SHA-256
- SHA-384
- SHA-512
- SHA256withRSA
- SHA384withRSA
- SHA512withRSA

- HmacMD5
- HmacSHA1
- HmacSHA256
- HmacSHA384
- HmacSHA512

Added the ability to pass in a configuration file to the provider. This configuration file can contain a significant amount of information about the device; for example, what it should or should not do. After the provider is created, the application can log in to the card in different ways. Some devices allow you to perform some cryptographic functions without logging into the device. The v1.4.2 ways to initialize the device still work. However, you can no longer have more than one of these providers at a time. Instead, with this release, you can initialize more than one IBMPKCS11Impl provider using the 5.0 configuration file and login methods.

Deprecated the classes DESPKCS11KeyParameterSpec and DESedePKCS11KeyParameterSpec. Use the GeneralPKCS11KeyParameterSpec class for all symmetric key types (for instance, DES, DESede, AES, RC4, Blowfish).

Changes for Version 1.4.2

The IBMPKCS11Impl Provider was new for v1.4.2.

IBMJCEFIPS Provider

The IBM Java JCE (Java Cryptographic Extension) FIPS Provider (IBMJCEFIPS) for multi-platforms is a scalable, multi-purpose cryptographic module that supports FIPS-approved cryptographic operations through Java APIs.

The IBMJCEFIPS includes the following Federal Information Processing Standards (FIPS) 140-2 [Level 1] compliant components:

- IBMJCEFIPS for Solaris
- IBMJCEFIPS for HP
- IBMJCEFIPS for Windows
- IBMJCEFIPS for z/OS
- IBMJCEFIPS for AS/400®
- IBMJCEFIPS for Linux (Red Hat and SUSE)

To meet the requirements specified in the FIPS publication 140-2, the encryption algorithms used by the IBMJCEFIPS Provider are isolated into the IBMJCEFIPS Provider cryptographic module. You can access the module using the product code from the Java JCE framework APIs. Because the IBMJCEFIPS Provider uses the cryptographic module in an approved manner, the product complies with the FIPS 140-2 requirements.

Type	Algorithm	Specification
Symmetric Cipher	AES (ECB, CBC, OFB, CFB, and PCBC)	FIPS 197
Symmetric Cipher	Triple DES (ECB, CBC, OFB, CFB, and PCBC)	FIPS 46-3

Type	Algorithm	Specification
Message Digest	SHA1 SHA-256 SHA-384 SHA-512 HMAC-SHA1	FIPS 180-2 FIPS 198a
Random Number Generator	FIPS 186-2 appendix 3.1	FIPS 186-2
Digital Signature	DSA (512 - 1024)	FIPS 186-2
Digital Signature	RSA (512 – 2048)	FIPS 186-2

In addition, the IBMJCEFIPS supports the following unapproved algorithms:

Type	Algorithm	Specification
Asymmetric Cipher	RSA	PKCS#1
Key Agreement	Diffie-Hellman	PKCS #3 (Allowed in Approved mode)
Digital Signature	DSAforSSL	Allowed for use inside the TLS protocol
Digital Signature	RSAforSSL	Allowed for use inside the TLS protocol
Message Digest	MD5	FIPS 180-2
Random Number Generation	Universal Software Based Random Number Generator	Available upon request from IBM. Patented by IBM, EC Pat. No. EP1081591A2, U.S. pat. Pend.

Important: The com.ibm.crypto.fips.provider.IBMJCEFIPS class does not include a keystore (such as JKS or JCEKS) because of FIPS requirements and algorithms. Therefore, if you are using com.ibm.crypto.fips.provider.IBMJCEFIPS and require JKS, you must specify the com.ibm.crypto.provider.IBMJCE in the provider list.

For more detailed information about the FIPS certified provider IBMJCEFIPS, see the *IBM Java JCE FIPS 140-2 Cryptographic Module Security Policy*. For usage information and details of the API, see the *IBM Java JCE FIPS (IBMJCEFIPS) Cryptographic Module API* document. These documents are available at http://www.ibm.com/developerworks/java/jdk/security/index.html.

Differences between IBM and Sun versions of IBMJCEFIPS

Sun does not provide IBMJCEFIPS.

History of changes

Version 1.4.2

IBMJCEFIPS is new for Version 1.4.2.

Documentation

For detailed information, including API documentation and Security Policy, see the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

IBM SASL Provider

Simple Authentication and Security Layer, or SASL, is an Internet standard (RFC 2222) that specifies a protocol for authentication and optional establishment of a security layer between client and server applications. SASL defines how authentication data is to be exchanged but does not itself specify the contents of that data. It is a framework into which specific authentication mechanisms that specify the contents and semantics of the authentication data can fit.

The Java SASL API defines classes and interfaces for applications that use SASL mechanisms. It is defined to be mechanism-neutral: the application that uses the API does not need to be hardwired into using any particular SASL mechanism. The API supports both client and server applications. It allows applications to select the mechanism to use based on preferred security features, such as whether they are susceptible to passive dictionary attacks or whether they accept anonymous authentication. The Java SASL API also allows developers to use their own, custom SASL mechanisms. SASL mechanisms are installed by using the Java Cryptography Architecture (JCA).

The IBMSASL provider supports the following client and server mechanisms.

Client mechanisms

- PLAIN (RFC 2595). This mechanism supports cleartext username/password authentication.
- CRAM-MD5 (RFC 2195). This mechanism supports a hashed username/password authentication scheme.
- DIGEST-MD5 (RFC 2831). This mechanism defines how HTTP Digest Authentication can be used as a SASL mechanism.
- GSSAPI (RFC 2222). This mechanism uses the GSSAPI for obtaining authentication information. It supports Kerberos v5 authentication.
- EXTERNAL (RFC 2222). This mechanism obtains authentication information from an external channel (such as TLS or IPsec).

Server mechanisms

- CRAM-MD5
- DIGEST-MD5
- GSSAPI (Kerberos v5)

Differences between the Sun and IBM SASL Providers

Only the package names, for example com.ibm.security.sasl, and the provider name are different from the Sun Implementation: com.ibm.security.sasl.IBMSASL.

History of changes

Version 5.0

The IBM SASL Provider is new for v5.0.

Documentation

Detailed information, including API documentation and samples, is on the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

Key Certificate Management utilities

Uses of the Key Certificate Management utilities.

The Key Certificate Management utilities make up a set of packages used to:

- Access keys and certificates stored in any format.
- Extract information from a KeyStore, given a Subject Key Identifier (SKI) and a set of certificate generation APIs, to create a self-signed certificate.
- Generate a CertificateRequest.
- Obtain a certificate signed by a CA.

The Key Certificate Management utilities can:

- Generate a CertificateRequest, and submit the request to a CA using the Java Public Key Infrastructure (PKI) to sign a certificate and then receive the signed certificate.
- Generate a PKCS10 request.
- Generate a Self-Signed Certificate.
- Revoke a signed certificate from a CA using the Java PKI.
- Import certificates from the input stream to the KeyStore or export certificates from the KeyStore to the output stream.
- Copy a keystore from one keystore format to another keystore format.
- Extract information from a KeyStore given a Subject Key Identifier.

The Subject Key Identifier is specified in RFC 3820, Section 4.2.1.2, http://www.faqs.org/rfcs/rfc3820.html.

History of changes

Version 5.0, Service Refresh 1
The Key Certificate Management utility is new for Version 5.0, Service Refresh 1.

Documentation

The *Key Certificate Management How-to Guide* and Javadoc information are on the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

Java XML Encryption and signatures

Java XML Encryption provides a standard set of APIs for XML digital encryption services and digital signature services.

You can use XML Encryption to perform fine-grained, element-based encryption of fragments in an XML Document. You can encrypt arbitrary binary data and include it in an XML document. The result of encrypting data is an XML Encryption element that contains or references the cipher data.

The IBM provider implementation of Java XML Encryption supports these algorithms:

- Block encryption: triple DES, AES-128 and AES-256
- Key transport: RSA-v1.5 and RSA-OAEP
- Symmetric key wrap: triple DES, AES-128 and AES-256

- Transformation: Base64, XPath, and XSLT

You can use XML Digital Signature to perform detached, enveloped, and enveloping signatures and to sign arbitrary binary data and include this signed data in an XML document. The result of encrypting data is an XML Signature element that contains or references the signature data.

The IBM provider implementation of Java XML Digital Signature supports these algorithms:

- Digest: SHA1
- Mac: HMAC-SHA1
- Signature: DSAwithSHA1, RSAwithSHA1
- Transformation: Canonicalization, Base64, XPath, and XSLT

History of changes

Version 6

Java XML Encryption and Java XML Digital Signature are new for Version 6.

Documentation

The user guide and Javadoc information are on the developerWorks Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1758
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.

1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

JIMMAIL@uk.ibm.com
[Hursley Java Technology Center (JTC) contact]

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of

performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Intel is a trademark of Intel Corporation in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product and service names may be trademarks or service marks of others.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

- JIMMAIL@uk.ibm.com [Hursley Java Technology Center (JTC) contact]

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel and Itanium are trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Printed in USA