WebSphere® IBM WebSphere Real Time V2 for AIX on 64-bit POWER

Version 2

User Guide

IBM®

WebSphere® IBM WebSphere Real Time V2 for AIX on 64-bit POWER
Version 2

User Guide

IBM®

Note

Before using this information and the product it supports, read the information in “Notices” on page 255.

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 AIX® on 64-bit POWER®.

Chapter 1. Introduction

This information center describes the IBM WebSphere Real Time for AIX on 64-bit POWER product referred to as WebSphere Real Time for AIX on 64-bit POWER in this information.

You can use this information to install and configure WebSphere Real Time for AIX on 64-bit POWER. Selected information about non-Real-Time Java™ is also provided here. See the Diagnostics Guide for further diagnostic information.

Viewing the information center

To use the information center on your local workstation, you must install the Eclipse Help system; see "Viewing the online help" on page 12. 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:

- System administrators who install and configure the Java environment; see Table 1.
- Service personnel team who maintain the Java environment; see Table 2.

Table 1. System administrators' tasks

Task	Reference
Planning for and overseeing product installation.	Chapter 2, "Installing IBM WebSphere Real Time for AIX on 64-bit POWER," on page 7

Table 2. Service personnel tasks

Task	Reference
Troubleshooting system and performance problems.	Chapter 5, "Troubleshooting OutOfMemory Errors," on page 25
Diagnosing problems.	"First steps in problem determination" on page 31

Overview of WebSphere Real Time for AIX on 64-bit POWER

WebSphere Real Time for AIX on 64-bit POWER bundles real-time capabilities with the standard JVM.

Features of WebSphere Real Time for AIX on 64-bit POWER

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

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 AIX on 64-bit POWER removes these obstacles by providing:
- The Metronome Garbage Collector, an incremental, deterministic garbage collector with very small pause times

What's new

This topic introduces anything new for IBM WebSphere Real Time for AIX on 64-bit POWER Refreshes

What's new for WebSphere Real Time for AIX on 64-bit POWER V2 Refresh 3

- 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 Metronome real time garbage collection technology.

Considerations

You must be aware of a number of factors when using WebSphere Real Time for AIX on 64-bit POWER.
- 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.
- When using shared class caches, the cache name must not exceed 53 characters.
- Workload partitions (WPAR) are not supported.
- Micropartitions are not supported. Logical partitions (LPAR) with an integral number of processors are supported, but LPARs with a fractional number of processors, for example 0.5 or 1.5, are not.
- Changed thread names.

  Some internal JVM thread names have changed in WebSphere Real Time for AIX on 64-bit POWER 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 thread. Similarly, the default name for a no-heap real-time thread is `NHRTThread-n`.

Performance considerations

WebSphere Real Time for AIX on 64-bit POWER is optimized for consistently short GC pauses rather than the highest throughput performance or smallest memory footprint.

Performance on certified hardware configurations

Certified systems have sufficient clock granularity and processor speed to support WebSphere Real Time for AIX on 64-bit POWER 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 of about 3 milliseconds, and no more than 3.2 milliseconds. During GC cycles, an application with default environment settings is not paused for more than 30% of elapsed time during any sliding 60 millisecond window. The collective time spent in GC pauses over any 60 millisecond period should add up to about 18 milliseconds.

Reducing timing variability

The main sources of variability in a standard JVM are garbage collection pauses. In WebSphere Real Time for AIX on 64-bit POWER, the potentially long pauses from standard Garbage Collector modes are avoided by using the Metronome Garbage Collector. See Chapter 3, "Using the Metronome Garbage Collector," on page 15.

Class data sharing between JVMs

Class data sharing provides a transparent method of reducing memory footprint and improving JVM start time. To learn more on class data sharing see "Class data sharing between JVMs"

Compressed references

From WebSphere Real Time for AIX on 64-bit POWER V2 SR3, the 64-bit JVM uses compressed references. When using compressed references, the JVM stores all references to objects, classes, threads, and monitors as 32-bit values. Using compressed references improves the performance of many applications because objects are smaller, resulting in less frequent garbage collection and improved memory cache utilization. For further information about compressed references, see the Memory Management section of the Diagnostics Guide.

Scheduling policies and priorities

From WebSphere Real Time for AIX on 64-bit POWER V2 SR3, regular Java threads can run with the policy SCHED_RR in addition to the default policy SCHED_OTHER. When running with the policy SCHED_RR, 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 Thread scheduling and dispatching.

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.

Class data sharing between JVMs

Support for shared classes is the same when running with, or without, the **-Xrealtime** option.

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*>
Limit the user and group to read and write access only for a particular cache	`chmod 660 /tmp/javasharedresources/`<*file for shared 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 170
A JVM can access a shared class cache with either read-write or read-only access. Read-write access is the default and gives all users equal rights to update the cache. Use the **-Xshareclasses:readonly** option for read-only access.

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.

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.

Chapter 2. Installing IBM WebSphere Real Time for AIX on 64-bit POWER

Follow these steps to install WebSphere Real Time for AIX on 64-bit POWER.

Hardware and software prerequisites

WebSphere Real Time for AIX on 64-bit POWER runs on AIX 5.3 TL 11 + SP 2, and on AIX 6.1 TL 4 + SP 2.

The latest service details and resources can be found here:http://www.ibm.com/developerworks/java/jdk/aix/service.html

To test whether WebSphere Real Time for AIX on 64-bit POWER is supported on a specific System p® system, at the system prompt type:

```
lscfg -p | fgrep Architecture
```

Supported platforms reply as follows:

```
Model Architecture: chrp
```

Only "Common Hardware Reference Platform" (CHRP) systems are supported by WebSphere Real Time for AIX on 64-bit POWER.

A minimum of 512 MB of physical memory is required for simple applications. For good performance, more complex applications require a larger memory configuration.

WebSphere Real Time for AIX on 64-bit POWER operates on Very Large Symmetric Multiprocessor systems. However, the additional computational power of systems with more than eight physical processor cores might give diminishing benefits. To optimize the extra capacity of these systems, multiple LPARs with up to eight physical processors each are recommended.

Installing WebSphere Real Time for AIX on 64-bit POWER

After downloading and unpacking the installation file you must complete some configuration tasks.

Before you begin

Ensure that the AIX operating system is correctly configured, and the required patches are installed. Details can be found here: "Hardware and software prerequisites." In particular, ensure you have installed the required APARs for your system.

If you installed a beta version of WebSphere Real Time for AIX on 64-bit POWER, follow the instructions in "Removing WebSphere Real Time for AIX on 64-bit POWER Beta code" on page 8 before proceeding.

Procedure

These steps need to be performed one time only:

Note: These instructions assume that you are working with the compressed runtime environment file or compressed SDK package file. In the steps outlined in this section the compressed file is called `ibm-wrt64-2.0.aix.tar.gz`. The file you download might have a different name, particularly if you are accessing Passport Advantage®.

1. Choose a suitable directory on your system, such as `$HOME/ibm-srt-sdk`
2. Copy the installation file to your chosen directory:
   ```
   cp ibm-wrt64-2.0.aix.tar.gz $HOME/ibm-srt-sdk
   ```
   The file is available from http://www.ibm.com/developerworks/java/jdk/aix/service.html
3. Change to the `$HOME/ibm-srt-sdk` directory:
   ```
   cd $HOME/ibm-srt-sdk
   ```
4. Extract the tar file using the following command:
   ```
   gunzip ibm-wrt64-2.0.aix.tar.gz
   ```
5. Extract the filesets from the tar file using the following command:
   ```
   tar xvf ibm-wrt64-2.0.aix.tar
   ```
6. Use the AIX installp command to install WebSphere Real Time for AIX on 64-bit POWER.
7. When the installation process is completed, you must change the user account to allow access to high resolution timers. Run the following command as root user:
   ```
   chuser "capabilities=CAP_NUMA_ATTACH,CAP_PROPAGATE" <username>
   ```
   where <*username*> is the non-root AIX user account.

   Note: This change needs to made to the user account only once. However, the user must log out and log back in for the change to take effect.

The following step must be completed in every shell before starting Java:

1. Set the **AIXTHREAD_HRT** environment variable to true. This environment variable allows a process to use high resolution time-outs with clock_nanosleep(). You must set this environment variable each time the process is started, using:
   ```
   AIXTHREAD_HRT=true
   ```
   This setting can be added to a user's `.profile` so that it is set each time the user logs in. Do this by adding the line:
   ```
   export AIXTHREAD_HRT=true
   ```

Removing WebSphere Real Time for AIX on 64-bit POWER Beta code

If you installed a beta version of IBM WebSphere Real Time for AIX on 64-bit POWER, remove the beta configuration changes before installing WebSphere Real Time for AIX on 64-bit POWER.

Removing the beta configuration changes

A number of changes are necessary to restore the original configuration settings. Follow these instructions to remove the changes that were made to access the realtime clock, and remove the workaround that changed the values for syscorepath.

Note: In these instructions, <*username*> refers to the non-root user who was originally granted the security privileges required to install WebSphere Real Time for AIX on 64-bit POWER.

1. Edit the `/etc/security/privcmds` file, and remove the following lines that were appended to the original file:

```
/bluebird/<username>/bin/platform/aix/bash:
 authprivs = realtime_auth=PV_PROC_RTCLK
 accessauths = ALLOW_ALL
 inheritprivs = PV_PROC_RTCLK
 secflags = FSF_EPS

/bluebird/<username>/bin/platform/aix/tcsh:
 authprivs = realtime_auth=PV_PROC_RTCLK
 accessauths = ALLOW_ALL
 inheritprivs = PV_PROC_RTCLK
 secflags = FSF_EPS

/usr/bin/sh:
 authprivs = realtime_auth=PV_PROC_RTCLK
 accessauths = ALLOW_ALL
 inheritprivs = PV_PROC_RTCLK
 secflags = FSF_EPS

/usr/bin/ksh:
 authprivs = realtime_auth=PV_PROC_RTCLK
 accessauths = ALLOW_ALL
 inheritprivs = PV_PROC_RTCLK
 secflags = FSF_EPS

/usr/bin/csh:
 authprivs = realtime_auth=PV_PROC_RTCLK
 accessauths = ALLOW_ALL
 inheritprivs = PV_PROC_RTCLK
 secflags = FSF_EPS
```

2. Edit the `/etc/security/user` file, and search for the non-root user account. Remove the following line:

```
default_roles = realtime_role
```

   The section containing this line is like the following:

```
<username>:
 admin = false
 capabilities = CAP_NUMA_ATTACH,CAP_PROPAGATE
 default_roles = realtime_role
```

   Note: <*username*> refers to the non-root user account.

3. Edit the `/etc/security/user.roles` file, and search for the non-root user account. Remove the following line:

```
default_roles = realtime_role
```

   The section containing this line is like the following:

```
<username>:
 roles = realtime_role
```

   Note: <*username*> refers to the non-root user account.

4. Rebuild the security tables by running the command:

```
setkst
```

5. Remove the `syscorepath` workaround that was applied, by typing the following command:

```
syscorepath -c
```

Relocating WebSphere Real Time for AIX on 64-bit POWER

By default, the WebSphere Real Time for AIX on 64-bit POWER SDK is installed in `/usr/java-ppc64-60-srt`. To install in another directory, use the AIX relocation commands.

Delete any `.toc` files in the directory containing your installp images or PTFs before using the AIX relocation commands.

Commands

See the AIX man pages for reference information about the command-line options for these commands.

installp_r

Install the SDK:

```
installp_r -a -Y -R /<Install Path>/ -d '.' WRT_64.sdk
```

Remove the SDK:

```
installp_r -u  -R /<Install Path>/ WRT_64.sdk
```

lsusil List the user-defined installation paths.

```
lsusil
```

lslpp_r

Find details of installed products.

```
lslpp_r -R /<Install Path>/ -S [A|O]
```

rmusil Remove existing user-defined installation paths.

```
rmusil -R /<Install Path>/
```

Verifying the installation

Follow these steps to check that your installation was successful.

Before you begin

To help ensure that the verification process behaves consistently, first run these commands:

```
unset LIBPATH
unset CLASSPATH
unset JAVA_COMPILER
unset JAVA_HOME
export PATH=/usr/java-ppc64-60-srt/jre/bin:/usr/java-ppc64-60-srt/bin:$PATH
```

Procedure

If you issue the command:

```
java -Xgcpolicy:metronome -version
```

you can expect to see this information, which verifies that the installation has worked:

```
java version "1.6.0"
Java(TM) SE Runtime Environment (build pap3260srtsr3-20090617_01(SR3))
IBM J9 VM (build 2.5, J2RE 1.6.0 IBM J9 2.5 AIX ppc-32 jvmap32srt60sr3-20090616_37328 (JIT enabled,
```

```
J9VM - 20090616_037328
JIT  - r10_20090615_2033
GC   - 20090603_AA)
JCL  - 20090603_01
```

Dates, times, and specific build information might be different.

What to do next

When verification is complete, log on again and review any values that you might have assigned to these variables for possible conflicts.

Setting the path

Updating the **PATH** environment variable enables AIX to find Java programs and utilities.

About this task

The **PATH** environment variable enables AIX to find programs and utilities, such as javac, java, and javadoc tool from any current directory. Changing the path will override any existing Java launchers in your path.

To display the current value of your **PATH** environment variable, 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. The name of your startup file will depend on the shell you are using; for example:
   - The Korn shell startup file is .kshrc.
   - The C shell startup file is .cshrc.
   - The Bourne shell startup file is .profile.
   - The BASH shell startup file is .bashrc.

   Add the absolute paths to the **PATH** environment variable; for example:

   ```
   export PATH=/usr/java-ppc64-60-srt/jre/bin:/usr/java-ppc64-60-srt/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 the tool command name at a command prompt from any directory. For example, to compile the file `Myfile.Java`, type:

```
javac Myfile.Java
```

Setting the classpath

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

About this task

Set the classpath explicitly only for these reasons:

- 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.

Viewing the online help

In the docs directory, the documentation is provided for use in the Eclipse Help System as com.ibm.softrt.aix64.doc.jar and com.ibm.softrt.aix64.doc.zip. The information is also provided as an Adobe® PDF file called softrt_aix64_jre.pdf.

About this task

- com.ibm.softrt.aix64.doc.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.softrt.aix64.doc.zip can be unpacked into the plug-in directory of your Eclipse Help System if the version is earlier than V3.1.1.
- softrt_aix64_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 AIX on 64-bit POWER Information Center to your Eclipse Help System.
   - **For Eclipse versions earlier than V3.1.1.** Extract the files from com.ibm.softrt.aix64.doc.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.softrt.aix64.doc.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 AIX on 64-bit POWER 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.

Chapter 3. Using the Metronome Garbage Collector

Metronome Garbage Collector replaces the standard Garbage Collector in WebSphere Real Time for AIX on 64-bit POWER.

Related reference

"Metronome Garbage Collector options" on page 247
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 -Xgcpolicy:metronome -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.

Metronome garbage collection and class unloading

Metronome supports class unloading in the same way as a standard Java developer kit. However, because of the work involved, while unloading classes there might be pause time outliers during garbage collection activities.

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, GC uses one thread for each logical active processor available to the operating system. This enables the most efficient parallel processing during GC cycles. A GC cycle means the time between GC being triggered and the completion of freeing garbage. Depending on the Java heap size, the elapsed time for a complete GC cycle could be several seconds. A GC cycle usually contains hundreds of GC quanta. These quanta are the very short pauses to application code, typically lasting 3 milliseconds. Use **-verbose:gc** to get summary reports of cycles and quanta. For more information, see: "Using verbose:gc information" on page 16. You can set the number of GC threads for the JVM using the **-Xgcthreads** option.

There is no benefit from increasing **-Xgcthreads** above the default. Reducing **-Xgcthreads** can reduce overall CPU load during GC cycles, though GC cycles will be lengthened.

Note: GC quanta duration targets remain constant at 3 milliseconds.

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
The collection threads perform the garbage 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 reference

"Metronome Garbage Collector options" on page 247
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 5, "Troubleshooting OutOfMemory Errors," on page 25
Dealing with OutOfMemoryError exceptions

Related information

"Tracing Java applications and the JVM" on page 126
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 -Xgcpolicy:metronome -verbose:gc -Xgc:verboseGCCycleTime=N myApplication
```

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="7" timestamp="Oct 20 20:42:49 2008" intervalms="317.456" />

<gc type="heartbeat" id="17" timestamp="Oct 20 20:42:50 2008" intervalms="1008.280">
  <summary quantumcount="51">
    <quantum minms="0.362" meanms="3.015" maxms="3.096" />
    <exclusiveaccess minms="0.003" meanms="0.014" maxms="0.044" />
    <heap minfree="532568668" meanfree="855123353" maxfree="1152691352" />
  </summary>
</gc>

<gc type="heartbeat" id="18" timestamp="Oct 20 20:42:51 2008" intervalms="1011.684">
  <summary quantumcount="54">
    <quantum minms="0.106" meanms="2.829" maxms="3.113" />
    <exclusiveaccess minms="0.003" meanms="0.018" maxms="0.055" />
    <heap minfree="410376144" meanfree="623433365" maxfree="818067840" />
  </summary>
</gc>

<gc type="heartbeat" id="19" timestamp="Oct 20 20:42:51 2008" intervalms="432.527">
  <summary quantumcount="21">
    <quantum minms="3.017" meanms="3.069" maxms="3.105" />
    <exclusiveaccess minms="0.004" meanms="0.025" maxms="0.053" />
    <classunloading classloaders="2" classes="1" />
    <refs_cleared soft="14" threshold="20" maxThreshold="32" weak="0" phantom="0" />
    <heap minfree="301004952" meanfree="420760698" maxfree="538701108" />
  </summary>
</gc>

<gc type="synchgc" id="6" timestamp="Oct 20 20:42:51 2008" intervalms="14.628">
  <details reason="out of memory" />
  <duration timems="185.741" />
  <heap freebytesbefore="290197120" />
  <heap freebytesafter="1462221448" />
</gc>

<gc type="trigger end" id="7" timestamp="Oct 20 20:42:51 2008" intervalms="2653.000" />
```

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 19

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.

<classunloading ...>
: The number of classloaders and classes unloaded during the heartbeat interval.

<refs_cleared ...>
: Is the number of Java reference objects that were cleared 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 20 20:42:27 2008" intervalms="0.055">
  <details reason="system garbage collect" />
  <duration timems="3.395" />
  <refs_cleared soft="0" threshold="31" maxThreshold="32" weak="2" phantom="0" />
  <finalization objectsqueued="2" />
  <heap freebytesbefore="2303633908" />
  <heap freebytesafter="2304483260" />
</gc>
```

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

```
<gc type="synchgc" id="1" timestamp="Oct 20 20:48:35 2008" intervalms="3.513">
  <details reason="vm shutdown" />
  <duration timems="3.295" />
  <refs_cleared soft="0" threshold="29" maxThreshold="32" weak="2" phantom="0" />
  <heap freebytesbefore="32765764" />
  <heap freebytesafter="62231696" />
</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.

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

<classunloading ...>
: The number of classloaders and classes unloaded during the heartbeat interval.

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

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=11
```

Out-of-memory entries

When the heap 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="Oct 20 21:03:19 2008" memoryspace="Metronome" J9MemorySpace="0x080B85B4" />
```

By default a Javadump is produced as a result of an OutOfMemoryError exception. This dump contains information about the memory used by your program.

```
0SECTION       MEMINFO subcomponent dump routine
NULL           =================================
NULL
1STMEMTYPE     Object Memory
NULL           region     start      end        size       name
1STHEAP        0x080B85B4 0xF4F20000 0xF6F10000 0x01FF0000 Default
NULL
```

```
1STMEMUSAGE    Total memory available: 33554432 (0x02000000)
1STMEMUSAGE    Total memory in use:    29396464 (0x01C08DF0)
1STMEMUSAGE    Total memory free:      04157968 (0x003F7210)
```

Related reference

"Metronome Garbage Collector options" on page 247
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 20 20:42:27 2008" intervalms="0.055">
  <details reason="system garbage collect" />
  <duration timems="3.395" />
  <refs_cleared soft="0" threshold="31" maxThreshold="32" weak="2" phantom="0" />
  <finalization objectsqueued="2" />
  <heap freebytesbefore="2303633908" />
  <heap freebytesafter="2304483260" />
</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.

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.

Chapter 4. The sample real-time hash map

The sample application uses a series of examples to demonstrate the features of WebSphere Real Time for AIX on 64-bit POWER 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 AIX on 64-bit POWER 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 AIX on 64-bit POWER 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 5. Troubleshooting OutOfMemory Errors

Dealing with OutOfMemoryError exceptions

Related concepts

"Troubleshooting the Metronome Garbage Collector" on page 16
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.

In general, a realtime 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 86.
- A Heapdump; see "Using Heapdump" on page 107.
- A Javadump; see "Using Javadump" on page 95

The dump file names are given in the console output:

The Java backtrace shown on the console output, and also available in the Javadump, indicates where in the Java application the OutOfMemoryError occurred. The JVM memory management component issues a tracepoint that gives the size, class block address, and memory space name of the failing allocation. This tracepoint can be found 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 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)

<< 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 1 on page 27 shows a view created by MDD4J detailing likely leak suspects, and giving the top five objects and packages contributing to the heap size.

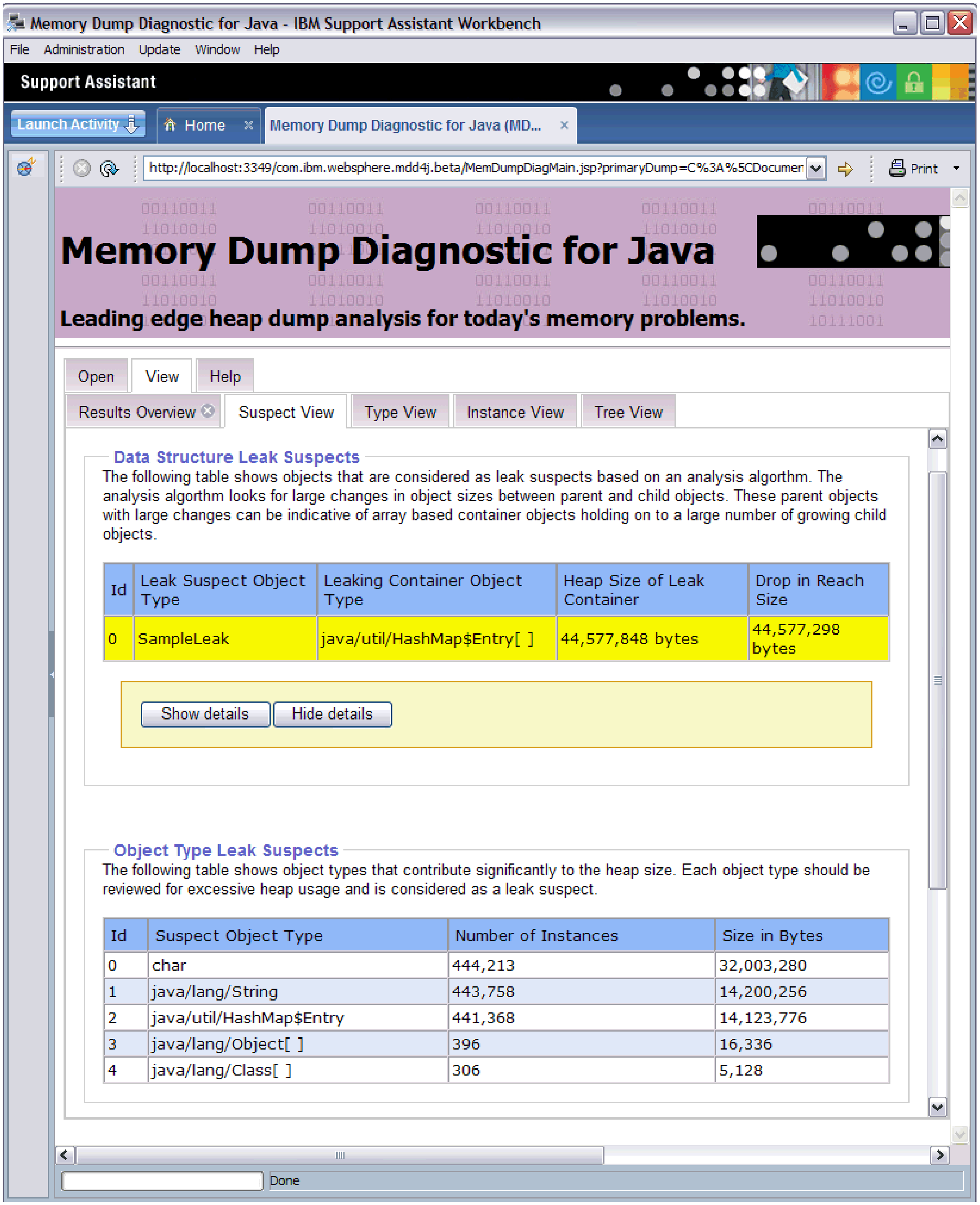


Figure 1. MDD4J has analyzed the heapdump and determined that there is a leak suspect

Selecting the tree view gives us further information about the nature of the leaking container object.

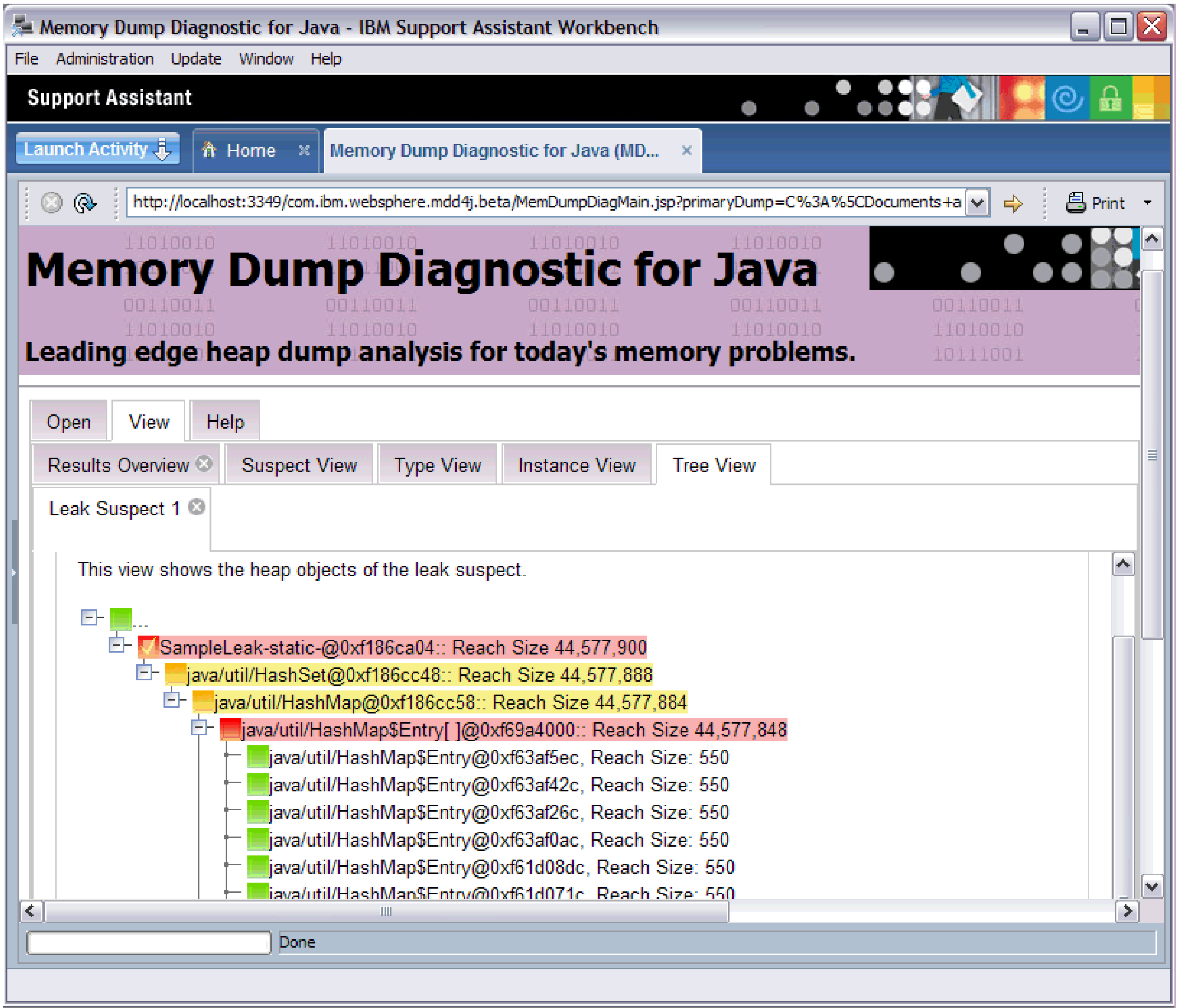


Figure 2. MDD4J shows the heap objects of the leak suspect

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, ROM classes and AOT code from the class cache are memory mapped, but not 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.

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
-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.

Use the **-Xshareclasses** option 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 `java -Xgcpolicy:metronome -Xshareclasses:printStats`.

```
Current statistics for cache "sharedcc_j9build":

shared memory ID = 48234504

base address = 0x070000002000DC0C
end address = 0x0700000021000000
allocation pointer = 0x07000000200F3394

cache size = 16776808
free bytes = 15267168
ROMClass bytes = 1412468
AOT bytes = 17728
Data bytes = 56504
Metadata bytes = 22940
Metadata % used = 1%

# ROMClasses = 365
# AOT Methods = 4
# Classpaths = 1
# URLs = 0
# Tokens = 0
# Stale classes = 0
% Stale classes = 0%

Cache is 8% full
```

Chapter 6. Problem determination

Problem determination helps you understand the kind of fault you have, and the appropriate course of action.

When you know what kind of problem you have, you might 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

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 “AIX problem determination”. 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 159.)

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 81 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 “AIX problem determination”.

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.

AIX problem determination

This section describes problem determination on AIX.

Setting up and checking your AIX environment

Set up the right environment for the AIX JVM to run correctly during AIX installation from either the installp image or the product with which it is packaged.

Note that the 64-bit JVM can work on a 32-bit kernel if the hardware is 64-bit. In that case, you must enable a 64-bit application environment using smitty:**System Environments -> Enable 64-bit Application Environment**.

Occasionally the configuration process does not work correctly, or the environment might be altered, affecting the operation of the JVM. In these conditions, you can make checks to ensure that the JVM's required settings are in place:

1. Check that the SDK and JRE files have been installed in the correct location and that the correct permissions are set. See the *User Guide* for more information about expected files and their location. Test the **java** and **javac** commands to ensure they are executable.

   The default installation directory is in `/usr/java6` for the 32-bit JVM and /usr/java6_64for the 64-bit JVM. For developer kits packaged with other products, the installation directory might be different; consult your product documentation.
2. Ensure that the **PATH** environment variable points to the correct Java executable (using `which java`), or that the application you are using is pointing to the correct Java directory. You must include the correct Java directory in your **PATH** environment variable . If it is not present, add it by using the appropriate `export PATH=</directory>:$PATH` statement; see "Setting the path" on page 11.
3. Ensure that the **LANG** environment variable is set to a supported locale. You can find the language environment in use using `echo $LANG`, which should report one of the supported locales as documented in the *User Guide* shipped with the SDK.
4. Ensure that all the prerequisite AIX maintenance and APARs have been installed. The prerequisite APARs and filesets will have been checked during an install using `smitty` or `installp`. You can find the list of prerequisites in the *User Guide* that is shipped with the SDK. Use `lslpp -l` to find the list of current filesets. Use `instfix -i -k <apar number>` to test for the presence of an APAR and `instfix -i | grep _ML` to find the installed maintenance level.

The **ReportEnv** tool, available from the Java service team, 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.

Directory requirements

The system dump agent must be configured to target a directory.

Both the user running the Java application and the group the user is in must have execute and write permissions for that directory. This can be set using the IBM_COREDIR environment variable.

The system dump agents can also be configured on the command line. See "Using dump agents" on page 81 for more information.

Enabling full AIX core files

You must have the correct operating system settings to ensure that the system dump (process core file) is generated when a failure occurs.

When a failure occurs, the most important diagnostic data to obtain is the system dump. The majority of the JVM settings are suitable by default but to ensure the system dump is generated on AIX, you must check a number of operating system settings.

If you do not enable full core dumps the only native thread details stored in the system dump are the details for the thread that was running when the JVM crashed. With full core dumps enabled, all native thread details are stored in the system dump.

Operating system settings

1. To obtain full system dumps, 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 -d unlimited   sets the user data limit to unlimited
   ulimit -f unlimited   sets the file limit to unlimited
   ```

   You can display the current `ulimit` settings with:

   ```
   ulimit -a
   ```

   These values are the "soft" limit, and are applied for each user. These values cannot exceed the "hard" limit value. To display and change the hard limits, you can run the ulimit commands using the additional **-H** command-line option.

   When the JVM generates a system dump it overrides the soft limit and uses the hard limit. You can disable the generation of system dumps by using the **-Xdump:system:none** command-line option.

2. Set the following in smitty:
   a. Start smitty as root
   b. Go to **System Environments** → **Change/Show Characteristics of Operating System**
   c. Set the **Enable full CORE dump** option to `TRUE`
   d. Ensure that the **Use pre-430 style CORE dump** option is set to `FALSE`

   Alternatively, you can run:

   ```
   chdev -l sys0 -a fullcore='true' -a pre430core='false'
   ```

   Note: Only the root user has r/w permission for the system core files created.

Note: System core files must be configured correctly. To do this, use the syscorepath command. For more information, see Chapter 2, "Installing IBM WebSphere Real Time for AIX on 64-bit POWER," on page 7.

Java Virtual Machine settings

The JVM settings should be in place by default, but you can check these settings using the following instructions.

To check that the JVM is set to produce a system dump when a failure occurs, run the following:

```
java -Xdump:what
```

which should include something like the following:

```
-Xdump:system:
    events=gpf+abort,
    label=/u/cbailey/core.%Y%m%d.%H%M%S.%pid.dmp,
    range=1..0,
    priority=999,
    request=serial
```

At least `events=gpf` must be set to generate a system dump when a failure occurs.

You can change and set options using the command-line option **-Xdump**, which is described in "Using dump agents" on page 81.

Available disk space
You must ensure that the disk space available is sufficient for the system dump to be written to it. The system dump is written to the directory specified in the label option. Up to 2 GB of free space might be required for 32-bit system dumps and over 6 GB for 64-bit system dumps. The Java process must have the correct permissions to write to the location specified in the label option.

General debugging techniques

A short guide to the diagnostic tools provided by the JVM and the AIX commands that can be useful when diagnosing problems with the AIX JVM.

In addition to the information in this section, you can obtain AIX publications from the IBM System p and AIX Information Center: http://www.ibm.com/servers/aix/. Of particular interest are:

- Performance management and tuning
- Programming for AIX

You might also find *"C and C++ Application Development on AIX" (SG24-5674)* helpful, available from: http://www.redbooks.ibm.com.

Action	Reference
Starting Javadumps	See "Using Javadump" on page 95.
Starting Heapdumps	See "Using Heapdump" on page 107.

AIX debugging commands

List of debugging commands.

bindprocessor –q
Lists the available processors.

bootinfo –K
Shows if the 64–bit kernel is active.

bootinfo –y
Shows whether the hardware in use is 32-bit or 64-bit.

dbx
The AIX debugger. Examples of use can be found throughout this set of topics.

The Java 5.0 SDK also includes a dbx Plug-in for additional help debugging Java applications. See "DBX Plug-in" on page 45 for more information.

iostat

Reports the read and write rate to all disks. This tool is useful in determining if you need to 'spread out' the disk workload across multiple disks. **iostat** also reports the same CPU activity that **vmstat** does.

lsattr
Details characteristics and values for devices in the system.

To obtain the type and speed of processor 0, use:

```
# lsattr -El proc0
state     enable         Processor state False
type      PowerPC_POWER3 Processor type  False
frequency 200000000      Processor Speed False
```

Processor 0 might not be available to you if you are using an LPAR. Use **bindprocessor -q** to list the available processors.

lsconf
Shows basic hardware and configuration details. See “lsconf” on page 37 for an example.

netpmon
uses the **trace** facility to obtain a detailed picture of network activity during a time interval. See “netpmon” on page 38 for an example.

netstat
Shows information about socket and network memory usage. Use this command with the **–m** option to look at mbuf memory usage. See “netstat” on page 39 for more details.

nmon
Gives much of the same information as topas, but saves the information to a file in Lotus® 123 and Excel formats.

The download site is http://www-941.haw.ibm.com/collaboration/wiki/display/WikiPtype/nmon. The information that is collected includes CPU, disk, network, adapter statistics, kernel counters, memory, and the 'top' process information.

no Configures network attributes. For example, to see the size of the wall use:

```
# no -a | grep wall
                  thewall = 524288
# no -o thewall =
1000000
```

The wall is the maximum amount of memory assigned to the network memory buffer.

ps Shows process information. See “ps” on page 40 for more details.

sar
Shows usage by multiple CPUs. See “sar” on page 42 for more details.

svmon
Captures snapshots of virtual memory. See “svmon” on page 42 for more details.

tprof
The **tprof** command reports CPU usage for individual programs and the system as a whole. This command is a useful tool for anyone with a Java program that might be CPU-bound and who wants to know which sections of the program are most heavily using the CPU.

The **tprof** command can charge CPU time to object files, processes, threads, subroutines (user mode, kernel mode and shared library) and even to source lines of programs or individual instructions. Charging CPU time to subroutines is called profiling and charging CPU time to source program lines is called micro-profiling.

topas
A graphical interface to system activity. See "topas" on page 44 for more details.

trace
Captures a sequential flow of time-stamped system events. The trace is a valuable tool for observing system and application execution. See "trace" on page 44 for more details.

truss
Traces a process's system calls, dynamically loaded user-level function calls, received signals, and incurred machine faults.

vmstat
Reports statistics about kernel threads in the run and wait queue, memory paging, interrupts, system calls, context switches, and CPU activity. See "vmstat" on page 44 for more details.

lsconf:

This command shows basic hardware and configuration details.

For example:

```
System Model: IBM,7040-681
Machine Serial Number: 835A7AA
Processor Type: PowerPC_POWER4
Number Of Processors: 8
Processor Clock Speed: 1100 MHz
CPU Type: 64-bit
Kernel Type: 64-bit
LPAR Info: 5 JAVADEV1 - kukicha
Memory Size: 10240 MB
Good Memory Size: 10240 MB
Platform Firmware level: 3H041021
Firmware Version: IBM,RG041021_d78e05_s
Console Login: enable
Auto Restart: true
Full Core: true

Network Information
        Host Name: bb1p5-1.hursley.ibm.com
        IP Address: 9.20.136.92
        Sub Netmask: 255.255.255.128
        Gateway: 9.20.136.1
        Name Server: 9.20.136.11
        Domain Name: hursley.ibm.com

Paging Space Information
        Total Paging Space: 512MB
        Percent Used: 21%

Volume Groups Information
==============================================================================
rootvg:
PV_NAME           PV STATE          TOTAL PPs   FREE PPs    FREE DISTRIBUTION
hdisk0            active            546         290         109..06..04..65..106
==============================================================================
```

```
INSTALLED RESOURCE LIST

The following resources are installed on the machine.
+/- = Added or deleted from Resource List.
*   = Diagnostic support not available.

  Model Architecture: chrp
  Model Implementation: Multiple Processor, PCI bus

+ sys0                           System Object
+ sysplanar0                     System Planar
* vio0                           Virtual I/O Bus
* vsa0                           LPAR Virtual Serial Adapter
* vty0                           Asynchronous Terminal
* pci12           U1.5-P2        PCI Bus
* pci11           U1.5-P2        PCI Bus
* pci10           U1.5-P2        PCI Bus
* pci9            U1.5-P1        PCI Bus
* pci14           U1.5-P1        PCI Bus
+ scsi0           U1.5-P1/Z2     Wide/Ultra-3 SCSI I/O Controller
+ hdisk0          U1.5-P1/Z2-A8  16 Bit LVD SCSI Disk Drive (73400 MB)
+ ses0            U1.5-P1/Z2-Af  SCSI Enclosure Services Device
* pci8            U1.5-P1        PCI Bus
* pci7            U1.5-P1        PCI Bus
* pci6            U1.9-P2        PCI Bus
* pci5            U1.9-P2        PCI Bus
* pci4            U1.9-P2        PCI Bus
* pci13           U1.9-P2        PCI Bus
+ ent0            U1.9-P2-I3/E1  Gigabit Ethernet-SX PCI Adapter (14100401)
* pci3            U1.9-P1        PCI Bus
* pci2            U1.9-P1        PCI Bus
* pci1            U1.9-P1        PCI Bus
* pci0            U1.18-P1-H2    PCI Bus
+ L2cache0                       L2 Cache
+ mem0                           Memory
+ proc11          U1.18-P1-C3    Processor
+ proc12          U1.18-P1-C3    Processor
+ proc13          U1.18-P1-C3    Processor
+ proc16          U1.18-P1-C4    Processor
+ proc17          U1.18-P1-C4    Processor
+ proc18          U1.18-P1-C4    Processor
+ proc22          U1.18-P1-C4    Processor
+ proc23          U1.18-P1-C4    Processor
```

netpmon:

This command uses the **trace** facility to obtain a detailed picture of network activity during a time interval.

It also displays process CPU statistics that show:
- The total amount of CPU time used by this process,
- The CPU usage for the process as a percentage of total time
- The total time that this process spent executing network-related code.

For example,

```
netpmon -o /tmp/netpmon.log; sleep 20; trcstop
```

is used to look for a number of things such as CPU usage by program, first level interrupt handler, network device driver statistics, and network statistics by program. Add the **-t** flag to produce thread level reports. The following output shows the processor view from netpmon.

```
Process CPU Usage Statistics:
-----------------------------
                                                   Network
Process (top 20)             PID  CPU Time   CPU %   CPU %
----------------------------------------------------------
java                       12192    2.0277   5.061   1.370
UNKNOWN                    13758    0.8588   2.144   0.000
gil                         1806    0.0699   0.174   0.174
UNKNOWN                    18136    0.0635   0.159   0.000
dtgreet                     3678    0.0376   0.094   0.000
swapper                        0    0.0138   0.034   0.000
trcstop                    18460    0.0121   0.030   0.000
sleep                      18458    0.0061   0.015   0.000
```

The adapter usage is shown here:

```
                        ----------- Xmit -----------   --------- Recv ---------
Device                   Pkts/s  Bytes/s  Util  QLen   Pkts/s  Bytes/s   Demux
------------------------------------------------------------------------------
token ring 0             288.95    22678  0.0%518.498   552.84    36761  0.0222
...
DEVICE: token ring 0
recv packets:           11074
  recv sizes (bytes):   avg 66.5    min 52      max 1514    sdev 15.1
  recv times (msec):    avg 0.008   min 0.005   max 0.029   sdev 0.001
  demux times (msec):   avg 0.040   min 0.009   max 0.650   sdev 0.028
xmit packets:           5788
  xmit sizes (bytes):   avg 78.5    min 62      max 1514    sdev 32.0
  xmit times (msec):    avg 1794.434 min 0.083   max 6443.266 sdev 2013.966
```

The following example shows the java extract:

```
 PROCESS: java   PID: 12192
reads:                  2700
  read sizes (bytes):   avg 8192.0  min 8192    max 8192    sdev 0.0
  read times (msec):    avg 184.061 min 12.430  max 2137.371 sdev 259.156
writes:                 3000
  write sizes (bytes):  avg 21.3    min 5       max 56      sdev 17.6
  write times (msec):   avg 0.081   min 0.054   max 11.426  sdev 0.211
```

To see a thread level report, add the `-t` as shown here.

```
netpmon -O so -t -o /tmp/netpmon_so_thread.txt; sleep 20; trcstop
```

The following extract shows the thread output:

```
        THREAD TID: 114559
reads:                  9
  read sizes (bytes):   avg 8192.0  min 8192    max 8192    sdev 0.0
  read times (msec):    avg 988.850 min 19.082  max 2106.933 sdev 810.518
writes:                 10
  write sizes (bytes):  avg 21.3    min 5       max 56      sdev 17.6
  write times (msec):   avg 0.389   min 0.059   max 3.321   sdev 0.977
```

You can also request that less information is gathered. For example to look at socket level traffic use the "-O so" option:

```
netpmon -O so -o /tmp/netpmon_so.txt; sleep 20; trcstop
```

netstat:

Use this command with the **–m** option to look at mbuf memory usage, which will tell you something about socket and network memory usage.

By default, the extended netstat statistics are turned off in `/etc/tc.net` with the line:

```
/usr/sbin/no -o extendednetstats=0 >>/dev/null 2>&1
```

To enable these statistics, change to `extendednetstats=1` and reboot. You can also try to set this directly with `no`. When using `netstat -m`, pipe to `page` because the first information is some of the most important:

```
67 mbufs in use:
64 mbuf cluster pages in use
272 Kbytes allocated to mbufs
0 requests for mbufs denied
0 calls to protocol drain routines
0 sockets not created because sockthresh was reached

-- At the end of the file:
Streams mblk statistic failures:
0 high priority mblk failures
0 medium priority mblk failures
0 low priority mblk failures
```

Use `netstat -i <interval to collect data>` to look at network usage and possible dropped packets.

ps:

Shows process information.

The Process Status (ps) is used to monitor:
- A process.
- Whether the process is still consuming CPU cycles.
- Which threads of a process are still running.

To start **ps** monitoring a process, type:

```
ps -fp <PID>
```

Your output should be:

```
 UID  PID     PPID     C    STIME    TTY    TIME    CMD
 user12  29730   27936    0    21 Jun   -      12:26   java StartCruise
```

Where

UID
The userid of the process owner. The login name is printed under the -f flag.

PPID
The Parent Process ID.

PID
The Process ID.

C
CPU utilization, incremented each time the system clock ticks and the process is found to be running. The value is decayed by the scheduler by dividing it by 2 every second. For the sched_other policy, CPU utilization is used in determining process scheduling priority. Large values indicate a CPU intensive process and result in lower process priority whereas small values indicate an I/O intensive process and result in a more favorable priority.

STIME
The start time of the process, given in hours, minutes, and seconds. The start time of a process begun more than twenty-four hours before the **ps** inquiry is executed is given in months and days.

TTY
: The controlling workstation for the process.

TIME
: The total execution time for the process.

CMD
: The full command name and its parameters.

To see which threads are still running, type:

```
ps -mp <PID> -o THREAD
```

Your output should be:

```
USER        PID   PPID      TID ST  CP PRI SC    WCHAN        F     TT BND COMMAND
user12    29730  27936        - A    4  60  8        *   200001 pts/10   0 java StartCruise
       -      -      -    31823 S    0  60  1 e6007cbc  8400400      -   0 -
       -      -      -    44183 S    0  60  1 e600acbc  8400400      -   0 -
       -      -      -    83405 S    2  60  1 50c72558   400400      -   0 -
       -      -      -   114071 S    0  60  1 e601bdbc  8400400      -   0 -
       -      -      -   116243 S    2  61  1 e601c6bc  8400400      -   0 -
       -      -      -   133137 S    0  60  1 e60208bc  8400400      -   0 -
       -      -      -   138275 S    0  60  1 e6021cbc  8400400      -   0 -
       -      -      -   140587 S    0  60  1 e60225bc  8400400      -   0 -
```

Where

USER
: The user name of the person running the process.

TID
: The Kernel Thread ID of each thread.

ST
: The state of the thread:

  O Nonexistent.

  R Running.

  S Sleeping.

  W Swapped.

  Z Canceled.

  T Stopped.

CP
: CPU utilization of the thread.

PRI
: Priority of the thread.

SC
: Suspend count.

ARCHON
: Wait channel.

F Flags.

TAT
: Controlling terminal.

BAND
: CPU to which thread is bound.

For more details, see the manual page for **ps**.

sar:

Use the **sar** command to check the balance of CPU usage for multiple CPU's.

In this following example, two samples are taken every five seconds on a 2-processor system that is 80% utilized.

```
# sar -u -P ALL 5 2

AIX aix4prt 0 5 000544144C00    02/09/01

15:29:32 cpu    %usr    %sys    %wio   %idle
15:29:37  0       34      46       0      20
          1       32      47       0      21
          -       33      47       0      20
15:29:42  0       31      48       0      21
          1       35      42       0      22
          -       33      45       0      22

Average   0       32      47       0      20
          1       34      45       0      22
          -       33      46       0      21
```

svmon:

This command captures snapshots of virtual memory. Using **svmon** to take snapshots of the memory usage of a process over regular intervals allows you to monitor memory usage.

The following usage of svmon generates regular snapshots of a process memory usage and writes the output to a file:

```
svmon -P [process id] -m -r -i [interval] > output.file
```

Gives output like:

```
     Pid Command        Inuse      Pin     Pgsp  Virtual   64-bit    Mthrd
   25084 AppS           78907     1570      182    67840        N        Y
    Vsid     Esid Type Description          Inuse    Pin Pgsp Virtual Addr Range
   2c7ea        3 work shmat/mmap           36678      0    0 36656   0..65513
   3c80e        4 work shmat/mmap            7956      0    0  7956   0..65515
   5cd36        5 work shmat/mmap            7946      0    0  7946   0..65517
   14e04        6 work shmat/mmap            7151      0    0  7151   0..65519
   7001c        d work shared library text   6781      0    0   736   0..65535
       0        0 work kernel seg            4218   1552  182  3602   0..22017 :
                                                                      65474..65535
   6cb5a        7 work shmat/mmap            2157      0    0  2157   0..65461
   48733        c work shmat/mmap            1244      0    0  1244   0..1243
    cac3        - pers /dev/hd2:176297       1159      0    -     -   0..1158
   54bb5        - pers /dev/hd2:176307        473      0    -     -   0..472
   78b9e        - pers /dev/hd2:176301        454      0    -     -   0..453
   58bb6        - pers /dev/hd2:176308        254      0    -     -   0..253
    cee2        - work                        246     17    0   246   0..49746
   4cbb3        - pers /dev/hd2:176305        226      0    -     -   0..225
   7881e        - pers /dev/e2axa702-1:2048   186      0    -     -   0..1856
   68f5b        - pers /dev/e2axa702-1:2048   185      0    -     -   0..1847
   28b8a        - pers /dev/hd2:176299        119      0    -     -   0..118
   108c4        - pers /dev/e2axa702-1:1843   109      0    -     -   0..1087
   24b68        f work shared library data     97      0    0    78   0..1470
   64bb9        - pers /dev/hd2:176311         93      0    -     -   0..92
   74bbd        - pers /dev/hd2:176315         68      0    -     -   0..67
   3082d        2 work process private         68      1    0    68   65287..65535
   10bc4        - pers /dev/hd2:176322         63      0    -     -   0..62
```

```
50815        1 pers code,/dev/hd2:210969      9     0    -     -   0..8
44bb1        - pers /dev/hd2:176303           7     0    -     -   0..6
7c83e        - pers /dev/e2axa702-1:2048      4     0    -     -   0..300
34a6c        a mmap mapped to sid 44ab0       0     0    -     -
70b3d        8 mmap mapped to sid 1c866       0     0    -     -
5cb36        b mmap mapped to sid 7cb5e       0     0    -     -
58b37        9 mmap mapped to sid 1cb66       0     0    -     -
1c7c7        - pers /dev/hd2:243801           0     0    -     -
```

in which:

Vsid
Segment ID

Esid
Segment ID: corresponds to virtual memory segment. The Esid maps to the Virtual Memory Manager segments. By understanding the memory model that is being used by the JVM, you can use these values to determine whether you are allocating or committing memory on the native or Java heap.

Type
Identifies the type of the segment:

pers Indicates a persistent segment.

work Indicates a working segment.

clnt Indicates a client segment.

mmap Indicates a mapped segment. This is memory allocated using mmap in a large memory model program.

Description

If the segment is a persistent segment, the device name and i-node number of the associated file are displayed.

If the segment is a persistent segment and is associated with a log, the string log is displayed.

If the segment is a working segment, the **svmon** command attempts to determine the role of the segment:

kernel
The segment is used by the kernel.

shared library
The segment is used for shared library text or data.

process private
Private data for the process.

shmat/mmap
Shared memory segments that are being used for process private data, because you are using a large memory model program.

Inuse
The number of pages in real memory from this segment.

Pin
The number of pages pinned from this segment.

Pgsp
The number of pages used on paging space by this segment. This value is relevant only for working segments.

Addr Range
: The range of pages that have been allocated in this segment. Addr Range displays the range of pages that have been allocated in each segment, whereas Inuse displays the number of pages that have been committed. For instance, **Addr Range** might detail more pages than **Inuse** because pages have been allocated that are not yet in use.

topas:

Topas is a useful graphical interface that will give you immediate information about system activity.

The screen looks like this:

```
Topas Monitor for host:    aix4prt                EVENTS/QUEUES    FILE/TTY
Mon Apr 16 16:16:50 2001   Interval:  2           Cswitch    5984  Readch     4864
                                                  Syscall   15776  Writech   34280
Kernel   63.1    |##################            | Reads         8  Rawin         0
User     36.8    |##########                    | Writes     2469  Ttyout        0
Wait      0.0    |                              | Forks         0  Igets         0
Idle      0.0    |                              | Execs         0  Namei         4
                                                  Runqueue   11.5  Dirblk        0
Network  KBPS   I-Pack  O-Pack   KB-In  KB-Out    Waitqueue   0.0
lo0     213.9   2154.2  2153.7   107.0   106.9
tr0      34.7     16.9    34.4     0.9    33.8    PAGING           MEMORY
                                                  Faults     3862  Real,MB    1023
Disk    Busy%     KBPS     TPS KB-Read KB-Writ    Steals     1580  % Comp     27.0
hdisk0    0.0      0.0     0.0     0.0     0.0    PgspIn        0  % Noncomp  73.9
                                                  PgspOut       0  % Client    0.5
Name         PID CPU% PgSp Owner                  PageIn        0
java       16684 83.6 35.1 root                   PageOut       0  PAGING SPACE
java       12192 12.7 86.2 root                   Sios          0  Size,MB     512
lrud        1032  2.7  0.0 root                                    % Used      1.2
aixterm    19502  0.5  0.7 root                   NFS (calls/sec)  % Free     98.7
topas       6908  0.5  0.8 root                   ServerV2       0
ksh        18148  0.0  0.7 root                   ClientV2       0   Press:
gil         1806  0.0  0.0 root                   ServerV3       0   "h" for help
```

trace:

This command captures a sequential flow of time-stamped system events. The trace is a valuable tool for observing system and application execution.

While many of the other tools provide high level statistics such as CPU and I/O utilization, the trace facility helps expand the information about where the events happened, which process is responsible, when the events took place, and how they are affecting the system. The **curt** postprocessing tool can extract information from the trace. It provides statistics on CPU utilization and process and thread activity. Another postprocessing tool is **splat**, the Simple Performance Lock Analysis Tool. This tool is used to analyze lock activity in the AIX kernel and kernel extensions for simple locks.

vmstat:

Use this command to give multiple statistics on the system. The **vmstat** command reports statistics about kernel threads in the run and wait queue, memory paging, interrupts, system calls, context switches, and CPU activity.

The CPU activity is percentage breakdown of user mode, system mode, idle time, and waits for disk I/O.

The general syntax of this command is:

```
vmstat  <time_between_samples_in_seconds> <number_of_samples> -t
```

A typical output looks like this:

```
kthr     memory             page              faults        cpu        time
----- ----------- ------------------------ ------------ ----------- --------
 r  b   avm   fre  re  pi  po  fr   sr  cy  in   sy  cs us sy id wa hr mi se
 0  0 45483   221   0   0   0   0    1   0 224  326 362 24  7 69  0 15:10:22
 0  0 45483   220   0   0   0   0    0   0 159   83  53  1  1 98  0 15:10:23
 2  0 45483   220   0   0   0   0    0   0 145  115  46  0  9 90  1 15:10:24
```

In this output, look for:

- Columns r (run queue) and b (blocked) starting to go up, especially above 10. This rise usually indicates that you have too many processes competing for CPU.
- Values in the pi, po (page in/out) columns at non-zero, possibly indicating that you are paging and need more memory. It might be possible that you have the stack size set too high for some of your JVM instances.
- cs (contact switches) going very high compared to the number of processes. You might have to tune the system with vmtune.
- In the cpu section, us (user time) indicating the time being spent in programs. Assuming Java is at the top of the list in tprof, you need to tune the Java application. In the cpu section, if sys (system time) is higher than expected, and you still have id (idle) time left, you might have lock contention. Check the tprof for lock–related calls in the kernel time. You might want to try multiple instances of the JVM.
- The **-t** flag, which adds the time for each sample at the end of the line.

DBX Plug-in

The Plug-in for the AIX DBX debugger gives DBX users enhanced features when working on Java processes or core files generated by Java processes.

The Plug-in requires a version of DBX that supports the Plug-in interface. Use the DBX command **pluginload** to find out whether your version of DBX has this support. All supported AIX versions include this support.

To enable the Plug-in, use the DBX command **pluginload**:

```
pluginload $JAVAHOME/jre/lib/ppc64/softrealtime/libdbx_j9.so
```

You can also set the **DBX_PLUGIN_PATH** environment variable to $JAVAHOME/jre/lib/ppc64/softrealtime. DBX automatically loads any Plug-ins found in the path given.

The commands available after loading the Plug-in can be listed by running:

```
plugin java help
```

from the DBX prompt.

You can also use DBX to debug your native JNI code by specifying the full path to the Java program as follows:

```
dbx $JAVAHOME/jre/lib/ppc64/softrealtime/java
```

Under DBX, issue the command:

```
(dbx) run <MyAppClass>
```

Before you start working with DBX, you must set the $java variable. Start DBX and use the `dbx set` subcommand. Setting this variable causes DBX to ignore the non-breakpoint traps generated by the JIT. You can also use a pre-edited command file by launching DBX with the `-c` option to specify the command file:

```
dbx -c .dbxinit
```

where `.dbxinit` is the default command file.

Although the DBX Plug-in is supplied as part of the SDK, it is not supported. However, IBM will accept bug reports.

Diagnosing crashes

If a crash occurs, you should gather some basic documents. These documents either point to the problem that is in the application or vendor package JNI code, or help the IBM JVM Support team to diagnose the fault.

A crash can occur because of a fault in the JVM or because of a fault in native (JNI) code being run in the Java process. Therefore, if the application does not include any JNI code and does not use any vendor-supplied packages that have JNI code (for example, JDBC application drivers), the fault must be in the JVM, and should be reported to IBM Support through the usual process.

Documents to gather

When a crash takes place, diagnostic data is required to help diagnose the problem.

- The output of stackit.sh run against the core file located as specified by the `label` field of **-Xdump:what**. stackit.sh is a script that runs a dbx session and is available from your Support Representative or from jvmcookbook@uk.ibm.com.
- The output of Jjextract run against the core file:

  ```
  jextract [core file]
  ```

  and collect the core.{date}.{time}.{pid}.dmp.zip output. See "Using system dumps and the dump viewer" on page 111 for details about jextract.
- Collect the javadump file. This file should be in the location detailed against the label field in **-Xdump:what** for javadumps
- Collect any stdout and stderr output generated by the Java process
- Collect the system error report:

  ```
  errpt -a > errpt.out
  ```

The above steps should leave you with the following files:

- stackit.out
- core.{date}.{time}.{pid}.dmp.zip
- javacore.{date}.{time}.{pid}.txt
- Snap<seq>.<date>.<time>.<pid>.trc
- errpt.out
- stderr/stdout files

Locating the point of failure

If a stack trace is present, examining the function running at the point of failure should give you a good indication of the code that caused the failure, and whether the failure is in IBM's JVM code, or is caused by application or vendor-supplied JNI code.

If dbx or stackit.sh produce no stack trace, the crash usually has two possible causes:

- A stack overflow of the native AIX stack.
- Java code is running (either JIT compiled or interpreted)

A failing instruction reported by dbx or stackit.sh as "stwu" indicates that there might have been a stack overflow. For example:

```
Segmentation fault in strlen at 0xd01733a0 ($t1)
0xd01733a0 (strlen+0x08) 88ac0000        stwu    r1,-80(r1)
```

You can check for the first cause by using the dbx command `thread info` and looking at the stack pointer, stack limit, and stack base values for the current thread. If the value of the stack pointer is close to that of the stack base, you might have had a stack overflow. A stack overflow occurs because the stack on AIX grows from the stack limit downwards towards the stack base. If the problem is a native stack overflow, you can solve the overflow by increasing the size of the native stack from the default size of 400K using the command-line option **-Xss<size>**. You are recommended always to check for a stack overflow, regardless of the failing instruction. To reduce the possibility of a JVM crash, you must set an appropriate native stack size when you run a Java program using a lot of native stack.

```
(dbx) thread info 1
 thread  state-k     wchan    state-u    k-tid   mode held scope function
>$t1     run                  running    85965     k   no   sys  oflow


      general:
         pthread addr = 0x302027e8         size         = 0x22c
         vp addr      = 0x302057e4         size         = 0x294
         thread errno = 0
         start pc     = 0x10001120
         joinable     = yes
         pthread_t    = 1
      scheduler:
         kernel       =
         user         = 1 (other)
      event :
         event        = 0x0
         cancel       = enabled, deferred, not pending
      stack storage:
         base         = 0x2df23000
    size         = 0x1fff7b0
          limit         = 0x2ff227b0
          sp           = 0x2df2cc70
```

For the second cause, currently dbx (and therefore stackit.sh) does not understand the structure of the JIT and Interpreter stack frames, and is not capable of generating a stack trace from them. The javadump, however, does not suffer from this limitation and can be used to examine the stack trace. A failure in JIT-compiled code can be verified and examined using the JIT Debugging Guide (see “JIT and AOT problem determination” on page 159).

Debugging hangs

The JVM is hanging if the process is still present but is not responding in some sense.

This lack of response can be caused because:

- The process has come to a complete halt because of a deadlock condition

- The process has become caught in an infinite loop
- The process is running very slowly

AIX deadlocks

If the process is not taking up any CPU time, it is deadlocked. Use the **ps -fp [process id]** command to investigate whether the process is still using CPU time.

The **ps** command is described in "AIX debugging commands" on page 35. For example:

```
$ ps -fp 30450
    UID   PID  PPID  C    STIME     TTY   TIME CMD
    root 30450 32332  2   15 May  pts/17  12:51 java ...
```

If the value of 'TIME' increases over the course of a few minutes, the process is still using the CPU and is not deadlocked.

For an explanation of deadlocks and how the Javadump tool is used to diagnose them, see "Locks, monitors, and deadlocks (LOCKS)" on page 101.

AIX busy hangs

If there is no deadlock between threads, consider other reasons why threads are not carrying out useful work.

Usually, this state occurs for one of the following reasons:

1. Threads are in a 'wait' state waiting to be 'notified' of work to be done.
2. Threads are in explicit sleep cycles.
3. Threads are in I/O calls waiting to do work.

The first two reasons imply a fault in the Java code, either that of the application, or that of the standard class files included in the SDK.

The third reason, where threads are waiting (for instance, on sockets) for I/O, requires further investigation. Has the process at the other end of the I/O failed? Do any network problems exist?

To see how the javadump tool is used to diagnose loops, see "Threads and stack trace (THREADS)" on page 102. If you cannot diagnose the problem from the javadump and if the process still seems to be using processor cycles, either it has entered an infinite loop or it is suffering from very bad performance. Using **ps -mp [process id] -o THREAD** allows individual threads in a particular process to be monitored to determine which threads are using the CPU time. If the process has entered an infinite loop, it is likely that a small number of threads will be using the time. For example:

```
$ ps -mp 43824 -o THREAD
    USER   PID  PPID     TID ST  CP PRI SC    WCHAN        F     TT BND COMMAND
  wsuser 43824 51762       - A   66  60 77        *   200001  pts/4   - java ...
       -     -     -    4021 S    0  60  1 22c4d670   c00400      -   - -
       -     -     -   11343 S    0  60  1 e6002cbc  8400400      -   - -
       -     -     -   14289 S    0  60  1 22c4d670   c00400      -   - -
       -     -     -   14379 S    0  60  1 22c4d670   c00400      -   - -
...
       -     -     -   43187 S    0  60  1 701e6114   400400      -   - -
       -     -     -   43939 R   33  76  1 20039c88   c00000      -   - -
       -     -     -   50275 S    0  60  1 22c4d670   c00400      -   - -
       -     -     -   52477 S    0  60  1 e600ccbc  8400400      -   - -
...
       -     -     -   98911 S    0  60  1 7023d46c   400400      -   - -
       -     -     -   99345 R   33  76  0        -   400000      -   - -
```

```
      -     -     -   99877 S    0  60   1 22c4d670   c00400      -   - -
      -     -     -  100661 S    0  60   1 22c4d670   c00400      -   - -
      -     -     -  102599 S    0  60   1 22c4d670   c00400      -   - -
...
```

Those threads with the value 'R' under 'ST' are in the 'runnable' state, and therefore are able to accumulate processor time. What are these threads doing? The output from **ps** shows the TID (Kernel Thread ID) for each thread. This can be mapped to the Java thread ID using **dbx**. The output of the dbx **thread** command gives an output of the form of:

```
thread  state-k     wchan    state-u    k-tid   mode held scope function
 $t1     wait      0xe60196bc blocked   104099     k   no   sys  _pthread_ksleep
>$t2     run                  blocked    68851     k   no   sys  _pthread_ksleep
 $t3     wait      0x2015a458 running    29871     k   no   sys  pthread_mutex_lock
...
 $t50    wait                 running    86077     k   no   sys  getLinkRegister
 $t51    run                  running    43939     u   no   sys  reverseHandle
 $t52    wait                 running    56273     k   no   sys  getLinkRegister
 $t53    wait                 running    37797     k   no   sys  getLinkRegister
 $t60    wait                 running     4021     k   no   sys  getLinkRegister
 $t61    wait                 running    18791     k   no   sys  getLinkRegister
 $t62    wait                 running    99345     k   no   sys  getLinkRegister
 $t63    wait                 running    20995     k   no   sys  getLinkRegister
```

By matching the TID value from **ps** to the **k-tid** value from the **dbx thread** command, you can see that the currently running methods in this case are reverseHandle and getLinkRegister.

Now you can use **dbx** to generate the C thread stack for these two threads using the dbx **thread** command for the corresponding dbx thread numbers (*$tx*). To obtain the full stack trace including Java frames, map the dbx thread number to the threads *pthread_t* value, which is listed by the Javadump file, and can be obtained from the ExecEnv structure for each thread using the Dump Viewer. Do this with the **dbx** command `thread info [dbx thread number]`, which produces an output of the form:

```
thread  state-k     wchan    state-u    k-tid   mode held scope function
 $t51    run                  running    43939     u   no   sys  reverseHandle
      general:
         pthread addr = 0x220c2dc0         size         = 0x18c
         vp addr      = 0x22109f94         size         = 0x284
         thread errno = 61
         start pc     = 0xf04b4e64
         joinable     = yes
         pthread_t    = 3233
      scheduler:
         kernel       =
         user         = 1 (other)
      event :
         event        = 0x0
         cancel       = enabled, deferred, not pending
      stack storage:
         base         = 0x220c8018         size         = 0x40000
         limit        = 0x22108018
         sp           = 0x22106930
```

Showing that the TID value from **ps** (**k-tid** in **dbx**) corresponds to dbx thread number 51, which has a *pthread_t* of 3233. Looking for the *pthread_t* in the Javadump file, you now have a full stack trace:

```
"Worker#31" (TID:0x36288b10, sys_thread_t:0x220c2db8) Native Thread State:
ThreadID: 00003233 Reuse: 1 USER SUSPENDED Native Stack Data : base: 22107f80
pointer 22106390 used(7152) free(250896)
----- Monitors held -----
```

```
java.io.OutputStreamWriter@3636a930
com.ibm.servlet.engine.webapp.BufferedWriter@3636be78
com.ibm.servlet.engine.webapp.WebAppRequestDispatcher@3636c270
com.ibm.servlet.engine.srt.SRTOutputStream@36941820
com.ibm.servlet.engine.oselistener.nativeEntry.NativeServerConnection@36d84490 JNI pinning lock

----- Native stack -----

_spin_lock_global_common pthread_mutex_lock - blocked on Heap Lock
sysMonitorEnterQuicker sysMonitorEnter unpin_object unpinObj
jni_ReleaseScalarArrayElements jni_ReleaseByteArrayElements
Java_com_ibm_servlet_engine_oselistener_nativeEntry_NativeServerConnection_nativeWrite

------ Java stack ------ () prio=5

com.ibm.servlet.engine.oselistener.nativeEntry.NativeServerConnection.write(Compiled Code)
com.ibm.servlet.engine.srp.SRPConnection.write(Compiled Code)
com.ibm.servlet.engine.srt.SRTOutputStream.write(Compiled Code)
java.io.OutputStreamWriter.flushBuffer(Compiled Code)
java.io.OutputStreamWriter.flush(Compiled Code)
java.io.PrintWriter.flush(Compiled Code)
com.ibm.servlet.engine.webapp.BufferedWriter.flushChars(Compiled Code)
com.ibm.servlet.engine.webapp.BufferedWriter.write(Compiled Code)
java.io.Writer.write(Compiled Code)
java.io.PrintWriter.write(Compiled Code)
java.io.PrintWriter.write(Compiled Code)
java.io.PrintWriter.print(Compiled Code)
java.io.PrintWriter.println(Compiled Code)
pagecompile._identifycustomer_xjsp.service(Compiled Code)
javax.servlet.http.HttpServlet.service(Compiled Code)
com.ibm.servlet.jsp.http.pagecompile.JSPState.service(Compiled Code)
com.ibm.servlet.jsp.http.pagecompile.PageCompileServlet.doService(Compiled Code)
com.ibm.servlet.jsp.http.pagecompile.PageCompileServlet.doGet(Compiled Code)
javax.servlet.http.HttpServlet.service(Compiled Code)
javax.servlet.http.HttpServlet.service(Compiled Code)
```

And, using the full stack trace, it should be possible to identify any infinite loop that might be occurring. The above example shows the use of `spin_lock_global_common`, which is a busy wait on a lock, hence the use of CPU time.

Poor performance on AIX

If no infinite loop is occurring, look at the process that is working, but having bad performance.

In this case, change your focus from what individual threads are doing to what the process as a whole is doing. This is described in the AIX documentation.

See “Debugging performance problems” on page 58 for more information about performance on AIX.

Understanding memory usage

Before you can properly diagnose memory problems on AIX, first you must have an understanding of the AIX virtual memory model and how the JVM interacts with it.

32- and 64-bit JVMs

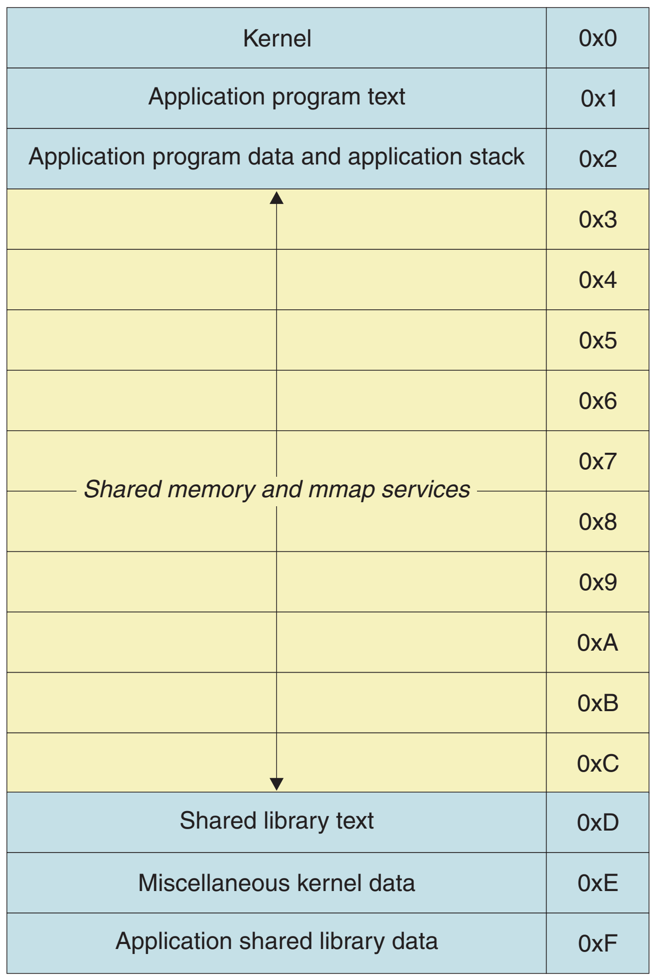
Most of the information in this section about altering the memory model and running out of native heap is relevant only to the 32-bit model, because the 64-bit model does not suffer from the same kind of memory constraints.

The 64-bit JVM can suffer from memory leaks in the native heap, and the same methods can be used to identify and pinpoint those leaks. The information regarding the Java heap relates to both 32- and 64-bit JVMs.

The 32-bit AIX Virtual Memory Model

AIX assigns a virtual address space partitioned into 16 segments of 256 MB.

Processing address space to data is managed at the segment level, so a data segment can either be shared (between processes), or private.



- Segment 0 is assigned to the kernel.
- Segment 1 is application program text (static native code).
- Segment 2 is the application program data and application stack (primordial thread stack and private data).
- Segments 3 to C are shared memory available to all processes.
- Segment D is the shared library text.
- Segment E is also shared memory and miscellaneous kernel usage.
- Segment F is the data area.

The 64-bit AIX Virtual Memory Model

The 64-bit model allows many more segments, although each segment is still 256 MB.

Again, the address space is managed at segment level, but the granularity of function for each segment is much finer.

With the large address space available to the 64-bit process, you are unlikely to encounter the same kind of problems with relation to native heap usage as described later in this section, although you might still suffer from a leak in the native heap.

Changing the Memory Model (32-bit JVM)

Three memory models are available on the 32-bit JVM.

Further details of the AIX Memory Models can be found at: http://publib.boulder.ibm.com/infocenter/pseries/v5r3/....

The small memory model

With the default small memory model for an application (as shown above), the application has only one segment, segment 2, in which it can `malloc()` data and allocate additional thread stacks. It does, however, have 11 segments of shared memory into which it can `mmap()` or `shmat()` data.

The large memory model

This single segment for data that is allocated by using `malloc()` might not be enough, so it is possible to move the boundary between Private and Shared memory, providing more Private memory to the application, but reducing the amount of Shared memory. You move the boundary by altering the *o_maxdata* setting in the Executable Common Object File Format (XCOFF) header for an application.

You can alter the *o_maxdata* setting by:

- Setting the value of *o_maxdata* at compile time by using the **-bmaxdata** flag with the **ld** command.
- Setting the *o_maxdata* value by using the **LDR_CNTRL=MAXDATA=0xn0000000 (n segments)** environment variable.

The very large memory model

Activate the very large memory model by adding "@DSA" onto the end of the **MAXDATA** setting. It provides two additional capabilities:

- The dynamic movement of the private and shared memory boundary between a single segment and the segment specified by the **MAXDATA** setting. This dynamic movement is achieved by allocating private memory upwards from segment 3 and shared memory downwards from segment C. The private memory area can expand upwards into a new segment if the segment is not being used by the shmat or mmap routines.
- The ability to load shared libraries into the process private area. If you specify a **MAXDATA** value of 0 or greater than 0xAFFFFFFF, the process will not use global shared libraries, but load them privately. Therefore, the shmat and mmap procedures begin allocating at higher segments because they are no longer reserved for shared libraries. In this way, the process has more contiguous memory.

Altering the **MAXDATA** setting applies only to a 32-bit process and not the 64-bit JVM.

The native and Java heaps

The JVM maintains two memory areas, the Java heap, and the native (or system) heap. These two heaps have different purposes and are maintained by different mechanisms.

The Java heap contains the instances of Java objects and is often referred to as 'the heap'. It is the Java heap that is maintained by Garbage Collection, and it is the Java heap that is changed by the command-line heap settings. The Java heap is allocated using mmap, or shmat if large page support is requested. The maximum size of the Java heap is preallocated during JVM startup as one contiguous area, even if the minimum heap size setting is lower. This allocation allows the artificial heap size limit imposed by the minimum heap size setting to move toward the actual heap size limit with heap expansion.

The native, or system heap, is allocated by using the underlying malloc and free mechanisms of the operating system, and is used for the underlying implementation of particular Java objects; for example:

- Motif objects required by AWT and Swing
- Buffers for data compression routines, which are the memory space that the Java Class Libraries require to read or write compressed data like .zip or .jar files.
- Malloc allocations by application JNI code
- Compiled code generated by the Just In Time (JIT) Compiler
- Threads to map to Java threads

The AIX 32-bit JVM default memory models

The AIX 5.0 Java launcher alters its **MAXDATA** setting in response to the command-line options to optimize the amount of memory available to the process. The default are as follows:

```
-Xmx <= 2304M 0xA0000000@DSA
2304M < -Xmx <= 3072M 0xB0000000@DSA
3072M < -Xmx 0x0@DSA
```

Monitoring the native heap

You can monitor the memory usage of a process by taking a series of snapshots over regular time intervals of the memory currently allocated and committed.

Use svmon like this:

```
svmon -P [pid] -m -r -i [interval] > output.filename
```

Use the `-r` flag to print the address range.

Because the Java heap is allocated using mmap() or shmat(), it is clear whether memory allocated to a specific segment of memory (under 'Esid') is allocated to the Java or the native heap. The type and description fields for each of the segments allows the determination of which sections are native or Java heap. Segments allocated using mmap or shmat are listed as "mmap mapped to" or "extended shm segments" and are the Java heap. Segments allocated using malloc will be marked as "working storage" and are in the native heap. This demarcation makes it possible to monitor the growth of the native heap separately from the Java heap (which should be monitored using verbose GC).

Here is the svmon output from the command that is shown above:

```
-------------------------------------------------------------------------------
     Pid Command          Inuse      Pin     Pgsp  Virtual 64-bit Mthrd LPage
   29670 java             87347     4782     5181    95830      N     Y     N
```

```
Vsid      Esid Type Description              LPage  Inuse   Pin Pgsp Virtual
50e9         - work                              -  41382     0    0 41382
               Addr Range: 0..41381
9dfb         - work                              -  28170     0 2550 30720
               Addr Range: 0..30719
ddf3         3 work working storage              -   9165     0  979 10140
               Addr Range: 0..16944
   0         0 work kernel seg                   -   5118  4766 1322  6420
               Addr Range: 0..11167
c819         d work text or shared-lib code seg  -   2038     0  283  6813
               Addr Range: 0..10219
2ded         f work working storage              -    191     0   20   224
               Addr Range: 0..4150
f5f6         - work                              -     41    14    4    45
               Addr Range: 0..49377
6e05         2 work process private              -     35     2   23    58
               Addr Range: 65296..65535
1140         6 work other segments               -     26     0    0    26
               Addr Range: 0..32780
cdf1         - work                              -      2     0    0     2
               Addr Range: 0..5277
e93f         - work                              -      0     0    0     0
3164         c mmap mapped to sid 1941           -      0     0    -     -
2166         - work                              -      0     0    0     0
496b         b mmap mapped to sid 2166           -      0     0    -     -
b51e         - clnt /dev/fslv00:44722            -      0     0    -     -
               Addr Range: 0..207
ee1c         a mmap mapped to sid e93f           -      0     0    -     -
1941         - work                              -      0     0    0     0
1081         7 mmap mapped to sid 9dfb           -      0     0    -     -
edf5         8 mmap mapped to sid 50e9           -      0     0    -     -
c01b         9 mmap mapped to sid cdf1           -      0     0    -     -
```

The actual memory values for the mmap allocated segments are stored against a Vsid of type "work". For example, the memory usage in segment 7 (Java heap):

```
1081         7 mmap mapped to sid 9dfb           -      0     0    -     -
```

is described against Vsid 9dfb, which reads as follows:

```
 9dfb         - work           -  28170     0 2550 30720   Addr Range: 0..30719
```

Native heap usage

The native heap usage will normally grow to a stable level, and then stay at around that level. You can monitor the amount of memory committed to the native heap by observing the number of 'Inuse' pages in the svmon output.

However, note that as JIT compiled code is allocated to the native heap with malloc(), there might be a steady slow increase in native heap usage as little used methods reach the threshold to undergo JIT compilation.

You can monitor the JIT compiling of code to avoid confusing this behavior with a memory leak. To do this, run with the command-line option **-Xjit:verbose={compileStart|compileEnd}**. This command causes each method name to print to stderr as it is being compiled and, as it finishes compiling, the location in memory where the compiled code is stored.

```
(warm) Compiling java/lang/System.getEncoding(I)Ljava/lang/String;
+ (warm) java/lang/System.getEncoding(I)Ljava/lang/String; @ 0x02BA0028-0x02BA0113
                                        (2) Compiling java/lang/String.hashCode()I
+ (warm) java/lang/String.hashCode()I @ 0x02BA0150-0x02BA0229
  (2) Compiling java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object;)
   Ljava/lang/Object;
+ (warm) java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object;)
```

```
                Ljava/lang/Object; @ 0x02BA0270-0x02BA03F7
               (2) Compiling java/lang/String.charAt(I)C
+ (warm) java/lang/String.charAt(I)C @ 0x02BA0430-0x02BA04AC
       (2) Compiling java/util/Locale.toLowerCase(Ljava/lang/String;)
               Ljava/lang/String;
+ (warm) java/util/Locale.toLowerCase(Ljava/lang/String;)Ljava/lang/String;
                                                @ 0x02BA04D0-0x02BA064C
```

When you have monitored how much native heap you are using, you can increase or decrease the maximum native heap available by altering the size of the Java heap. This relationship between the heaps occurs because the process address space not used by the Java heap is available for the native heap usage.

You must increase the native heap if the process is generating errors relating to a failure to allocate native resources or exhaustion of process address space. These errors can take the form of a JVM internal error message or a detail message associated with an OutOfMemoryError. The message associated with the relevant errors will make it clear that the problem is native heap exhaustion.

Specifying MALLOCTYPE

You can set the **MALLOCTYPE=watson** environment variable, available in AIX 5.3, for use with the IBM 5.0 JVM, but for most applications the performance gains are likely to be small. It particularly benefits any application that makes heavy use of malloc calls in native code.

For more information, see this article: http://www.ibm.com/developerworks/eserver/library/es-appdev-aix5l.html.

Monitoring the Java heap

The most straightforward, and often most useful, way of monitoring the Java heap is by seeing what garbage collection is doing.

Start verbose tracing of garbage collection by using the command-line option **-verbose:gc**. The option causes a report to be written to `stderr` each time garbage collection occurs. You can also direct this output to a log file using:

```
-Xverbosegclog:[DIR_PATH][FILE_NAME]
```

where:

```
[DIR_PATH]   is the directory where the file should be written
[FILE_NAME]   is the name of the file to write the logging to
```

See "Garbage Collector diagnostics" on page 169 for more information about verbose GC output and monitoring.

Receiving OutOfMemoryError exceptions

An OutOfMemoryError exception results from running out of space on the Java heap or the native heap.

If the process address space (that is, the native heap) is exhausted, an error message is received that explains that a native allocation has failed. In either case, the problem might not be a memory leak, just that the steady state of memory use that is required is higher than that available. Therefore, the first step is to determine which heap is being exhausted and increase the size of that heap.

If the problem is occurring because of a real memory leak, increasing the heap size does not solve the problem, but does delay the onset of the OutOfMemoryError exception or error conditions. That delay can be helpful on production systems.

The maximum size of an object that can be allocated is limited only by available memory. The maximum number of array elements supported is 2^31 - 1, the maximum permitted by the Java Virtual Machine specification. In practice, you might not be able to allocate large arrays due to available memory. Configure the total amount of memory available for objects using the **-Xmx** command-line option.

These limits apply to both 32-bit and 64-bit JVMs.

Is the Java or native heap exhausted?

Some OutOfMemory conditions also carry an explanatory message, including an error code.

If a received OutOfMemory condition has one of these codes or messages, consulting the Diagnosis Guide might point to the origin of the error, either native or Java heap.

If no error message is present, the first stage is to monitor the Java and native heap usages. The Java heap usage can be monitored by using the **-verbose:gc** option. The native heap can be monitored using svmon.

Java heap exhaustion

The Java heap becomes exhausted when garbage collection cannot free enough objects to make a new object allocation.

Garbage collection can free only objects that are no longer referenced by other objects, or are referenced from the thread stacks (see Memory management for more details).

Java heap exhaustion can be identified from the **-verbose:gc** output by garbage collection occurring more and more frequently, with less memory being freed. Eventually the JVM will fail, and the heap occupancy will be at, or almost at, 100% (See Memory management for more details on **-verbose:gc** output).

If the Java heap is being exhausted, and increasing the Java heap size does not solve the problem, the next stage is to examine the objects that are on the heap, and look for suspect data structures that are referencing large numbers of Java objects that should have been released. Use Heapdump Analysis, as detailed in "Using Heapdump" on page 107. Similar information can be gained by using other tools, such as JProbe and OptimizeIt.

Native heap exhaustion

You can identify native heap exhaustion by monitoring the svmon snapshot output

Each segment is 256 MB of space, which corresponds to 65535 pages. (Inuse is measured in 4 KB pages.)

If each of the segments has approximately 65535 Inuse pages, the process is suffering from native heap exhaustion. At this point, extending the native heap size might solve the problem, but you should investigate the memory usage profile to ensure that you do not have a leak.

If DB2 is running on your AIX system, you can change the application code to use the "net" (thin client) drivers and, in the case of WebSphere MQ you can use the "client" (out of process) drivers.

AIX fragmentation problems

Native heap exhaustion can also occur without the Inuse pages approaching 65535 Inuse pages. It can be caused by fragmentation of the AIX malloc heaps, which is how AIX handles the native heap of the JVM.

This kind of OutOfMemory condition can again be identified from the svmon snapshots. Whereas previously the important column to look at for a memory leak is the Inuse values, for problems in the AIX malloc heaps it is important to look at the 'Addr Range' column. The 'Addr Range' column details the pages that have been allocated, whereas the Inuse column details the number of pages that are being used (committed).

It is possible that pages that have been allocated have not been released back to the process when they have been freed. This leads to the discrepancy between the number of allocated and committed pages.

You have a range of environment variables to change the behavior of the malloc algorithm itself and thereby solve problems of this type:

MALLOCTYPE=3.1
: This option allows the system to move back to an older version of memory allocation scheme in which memory allocation is done in powers of 2. The 3.1 Malloc allocator, as opposed to the default algorithm, frees pages of memory back to the system for reuse. The 3.1 allocation policy is available for use only with 32-bit applications.

MALLOCMULTIHEAP=heaps:n,considersize
: By default, the malloc subsystem uses a single heap. **MALLOCMULTIHEAP** allows users to enable the use of multiple heaps of memory. Multiple heaps of memory can lead to memory fragmentation, and so the use of this environment variable is not recommended

MALLOCTYPE=buckets
: Malloc buckets provides an optional buckets-based extension of the default allocator. It is intended to improve malloc performance for applications that issue large numbers of small allocation requests. When malloc buckets is enabled, allocation requests that fall inside a predefined range of block sizes are processed by malloc buckets. Because of variations in memory requirements and usage, some applications might not benefit from the memory allocation scheme used by malloc buckets. Therefore, it is not advisable to enable malloc buckets system-wide. For optimal performance, enable and configure malloc buckets on a per-application basis.

  Note: The above options might cause a percentage of performance hit. Also the 3.1 malloc allocator does not support the Malloc Multiheap and Malloc Buckets options.

MALLOCBUCKETS= number_of_buckets:128,bucket_sizing_factor:64,blocks_per_bucket:1024: bucket_statistics: <path name of file for malloc statistics>
: See above.

Submitting a bug report

If the data is indicating a memory leak in native JVM code, contact the IBM service team. If the problem is Java heap exhaustion, it is much less likely to be an SDK issue, although it is still possible.

The process for raising a bug is detailed in Submitting problem reports, and the data that should be included in the bug report is listed as follows:

- Required:
  1. The OutOfMemoryCondition. The error itself with any message or stack trace that accompanied it.
  2. **-verbose:gc** output. (Even if the problem is determined to be native heap exhaustion, it can be useful to see the verbose gc output.)
- As appropriate:
  1. The svmon snapshot output
  2. The Heapdump output
  3. The javacore.txt file

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

The aspects of the system that you are most interested in measuring are CPU usage and memory usage.

It is possible that even after extensive tuning efforts the CPU is not powerful enough to handle the workload, in which case a CPU upgrade is required. Similarly, if the program is running in an environment in which it does not have enough memory after tuning, you must increase memory size.

Given that any performance problem could be caused by any one of several factors, you must look at several areas to eliminate each one. First, determine which resource is constraining the system:

- CPU
- Memory
- Input/Output (I/O)

To do this, use the **vmstat** command. The **vmstat** command produces a compact report that details the activity of these three areas:

```
> vmstat 1 10
```

outputs:

```
kthr     memory             page              faults        cpu
----- ----------- ------------------------ ------------ -----------
 r  b   avm   fre  re  pi  po  fr   sr  cy  in   sy  cs us sy id wa
 0  0 189898   612   0   0   0   3   11   0 178  606 424  6  1 92  1
 1  0 189898   611   0   1   0   0    0   0 114 4573 122 96  4  0  0
 1  0 189898   611   0   0   0   0    0   0 115  420 102 99  0  0  0
 1  0 189898   611   0   0   0   0    0   0 115  425  91 99  0  0  0
 1  0 189898   611   0   0   0   0    0   0 114  428  90 99  0  0  0
 1  0 189898   610   0   1   0   0    0   0 117  333 102 97  3  0  0
 1  0 189898   610   0   0   0   0    0   0 114  433  91 99  1  0  0
```

```
 1  0 189898   610   0   0   0   0    0   0 114  429  94 99  1  0  0
 1  0 189898   610   0   0   0   0    0   0 115  437  94 99  0  0  0
 1  0 189898   609   0   1   0   0    0   0 116  340  99 98  2  0  0
```

The example above shows a system that is CPU bound. This can be seen as the user (us) plus system (sy) CPU values either equal or are approaching 100. A system that is memory bound shows significant values of page in (pi) and page out (po). A system that is disk I/O bound will show an I/O wait percentage (wa) exceeding 10%. More details of vmstat can be found in "AIX debugging commands" on page 35.

CPU bottlenecks

If vmstat has shown that the system is CPU-bound, the next stage is to determine which process is using the most CPU time.

The recommended tool is tprof:

```
> tprof -s -k -x sleep 60
```

outputs:

```
Mon Nov 28 12:40:11 2005
System: AIX 5.2 Node: voodoo Machine: 00455F1B4C00

Starting Command sleep 60
stopping trace collection
Generating sleep.prof
> cat sleep.prof
Process                               Freq  Total Kernel   User Shared  Other
=======                               ====  ===== ======   ==== ======  =====
./java                                   5  59.39  24.28   0.00  35.11   0.00
wait                                     4  40.33  40.33   0.00   0.00   0.00
/usr/bin/tprof                           1   0.20   0.02   0.00   0.18   0.00
/etc/syncd                               3   0.05   0.05   0.00   0.00   0.00
/usr/bin/sh                              2   0.01   0.00   0.00   0.00   0.00
gil                                      2   0.01   0.01   0.00   0.00   0.00
afsd                                     1   0.00   0.00   0.00   0.00   0.00
rpc.lockd                                1   0.00   0.00   0.00   0.00   0.00
swapper                                  1   0.00   0.00   0.00   0.00   0.00
=======                               ====  ===== ======   ==== ======  =====
Total                                   20 100.00  64.70   0.00  35.29   0.00

Process                    PID      TID  Total Kernel   User Shared  Other
=======                    ===      ===  ===== ======   ==== ======  =====
./java                  467018   819317  16.68   5.55   0.00  11.13   0.00
./java                  467018   766019  14.30   6.30   0.00   8.00   0.00
./java                  467018   725211  14.28   6.24   0.00   8.04   0.00
./java                  467018   712827  14.11   6.16   0.00   7.94   0.00
wait                     20490    20491  10.24  10.24   0.00   0.00   0.00
wait                      8196     8197  10.19  10.19   0.00   0.00   0.00
wait                     12294    12295   9.98   9.98   0.00   0.00   0.00
wait                     16392    16393   9.92   9.92   0.00   0.00   0.00
/usr/bin/tprof          421984   917717   0.20   0.02   0.00   0.18   0.00
/etc/syncd              118882   204949   0.04   0.04   0.00   0.00   0.00
./java                  467018   843785   0.03   0.02   0.00   0.00   0.00
gil                      53274    73765   0.00   0.00   0.00   0.00   0.00
gil                      53274    61471   0.00   0.00   0.00   0.00   0.00
/usr/bin/sh             397320   839883   0.00   0.00   0.00   0.00   0.00
rpc.lockd               249982   434389   0.00   0.00   0.00   0.00   0.00
/usr/bin/sh             397318   839881   0.00   0.00   0.00   0.00   0.00
swapper                      0        3   0.00   0.00   0.00   0.00   0.00
afsd                     65776   274495   0.00   0.00   0.00   0.00   0.00
/etc/syncd              118882   258175   0.00   0.00   0.00   0.00   0.00
/etc/syncd              118882   196839   0.00   0.00   0.00   0.00   0.00
=======                    ===      ===  ===== ======   ==== ======  =====
```

```
Total                                  100.00  64.70   0.00  35.29   0.00

 Total Samples = 24749 Total Elapsed Time = 61.88s
```

This output shows that the Java process with Process ID (PID) 467018 is using the majority of the CPU time. You can also see that the CPU time is being shared among four threads inside that process (Thread IDs 819317, 766019, 725211, and 712827).

By understanding what the columns represent, you can gather an understanding of what these threads are doing:

Total
The total percentage of CPU time used by this thread or process.

Kernel
The total percentage of CPU time spent by this thread or process inside Kernel routines (on behalf of a request by the JVM or other native code).

User
The total percentage of CPU time spent executing routines inside the executable. Because the Java executable is a thin wrapper that loads the JVM from shared libraries, this CPU time is expected to be very small or zero.

Shared
The total percentage of CPU time spent executing routines inside shared libraries. Time shown under this category covers work done by the JVM itself, the act of JIT compiling (but not the running of the subsequent code), and any other native JNI code.

Other
The total percentage of CPU time not covered by Kernel, User, and Shared. In the case of a Java process, this CPU time covers the execution of Java bytecodes and JIT-compiled methods themselves.

From the above example, notice the Kernel and Shared values: these account for all of the CPU time used by this process, indicating that the Java process is spending its time doing work inside the JVM (or some other native code).

To understand what is being done during the Kernel and Shared times, the relevant sections of the tprof output can be analyzed.

The shared library section shows which shared libraries are being invoked:

```
Shared Object                                                          %
=============                                                     ======
/j9vmap3224-20071224/inst.images/rios_aix32_6/sdk/jre/bin/libj9gc24.so  17.42
/usr/lib/libc.a[shr.o]                                                   9.38
/usr/lib/libpthreads.a[shr_xpg5.o]                                       6.94
j9vmap3224-20071224/inst.images/rios_aix32_6/sdk/jre/bin/libj9thr24.so   1.03
j9vmap3224-20071224/inst.images/rios_aix32_6/sdk/jre/bin/libj9prt24.so   0.24
/j9vmap3224-20071224/inst.images/rios_aix32_6/sdk/jre/bin/libj9vm24.so   0.10
j9vmap3224-20071224/inst.images/rios_aix32_6/sdk/jre/bin/libj9ute24.so   0.06
j9vmap3224-20071224/inst.images/rios_aix32_6/sdk/jre/bin/libj9jit24.so   0.05
/usr/lib/libtrace.a[shr.o]                                               0.04
j9vmap3224-20071224/inst.images/rios_aix32_6/sdk/jre/bin/libj9trc24.so   0.02
p3224-20071224/inst.images/rios_aix32_6/sdk/jre/bin/libj9hookable24.so   0.01
```

This section shows that almost all of the time is being spent in one particular shared library, which is part of the JVM installation: libj9gc24.so. By understanding the functions that the more commonly used JVM libraries carry out, it becomes possible to build a more accurate picture of what the threads are doing:

libbcv24.so
: Bytecode Verifier

libdbg24.so
: Debug Server (used by the Java Debug Interface)

libj9gc24.so
: Garbage Collection

libj9jextract.so
: The dump extractor, used by the jextract command

libj9jit24.so
: The Just In Time (JIT) Compiler

libj9jvmti24.so
: The JVMTI interface

libj9prt24.so
: The "port layer" between the JVM and the Operating System

libj9shr24.so
: The shared classes library

libj9thr24.so
: The threading library

libj9ute24.so
: The trace engine

libj9vm24.so
: The core Virtual Machine

libj9zlib24.so
: The compressed file utility library

libjclscar_24.so
: The Java Class Library (JCL) support routines

In the example above, the CPU time is being spent inside the garbage collection (GC) implementation, implying either that there is a problem in GC or that GC is running almost continuously.

Again, you can obtain a more accurate understanding of what is occurring inside the `libj9gc24.so` library during the CPU time by analyzing the relevant section of the tprof output:

```
Profile: /work/j9vmap3224-20071224/inst.images/rios_aix32_6/sdk/jre/bin/
                                                      libj9gc24.so

  Total % For All Processes (/work/j9vmap3224-20071224/inst.images/rios_aix32_6/
                                 sdk/jre/bin/libj9gc24.so) = 17.42

Subroutine                                             %   Source
==========                                          ====== ======
Scheme::scanMixedObject(MM_Environment*,J9Object*)   2.67 MarkingScheme.cpp
MarkingScheme::scanClass(MM_Environment*,J9Class*)   2.54 MarkingScheme.cpp
.GC_ConstantPoolObjectSlotIterator::nextSlot()       1.96 jectSlotIterator.cpp
lelTask::handleNextWorkUnit(MM_EnvironmentModron*)   1.05 ParallelTask.cpp
orkPackets::getPacket(MM_Environment*,MM_Packet**)   0.70 WorkPackets.cpp
cheme::fixupRegion(J9Object*,J9Object*,bool,long&)   0.67 CompactScheme.cpp
WorkPackets::putPacket(MM_Environment*,MM_Packet*)   0.47 WorkPackets.cpp
rkingScheme::scanObject(MM_Environment*,J9Object*)   0.43 MarkingScheme.cpp
sweepChunk(MM_Environment*,MM_ParallelSweepChunk*)   0.42 allelSweepScheme.cpp
ment*,J9IndexableObject*,J9Object**,unsigned long)   0.38 MarkingScheme.cpp
M_CompactScheme::getForwardingPtr(J9Object*) const   0.36 CompactScheme.cpp
```

```
ObjectHeapIteratorAddressOrderedList::nextObject()   0.33 dressOrderedList.cpp
ckets::getInputPacketFromOverflow(MM_Environment*)   0.32 WorkPackets.cpp
.MM_WorkStack::popNoWait(MM_Environment*)            0.31 WorkStack.cpp
WorkPackets::getInputPacketNoWait(MM_Environment*)   0.29 WorkPackets.cpp
canReferenceMixedObject(MM_Environment*,J9Object*)   0.29 MarkingScheme.cpp
MarkingScheme::markClass(MM_Environment*,J9Class*)   0.27 MarkingScheme.cpp
._ptrgl                                              0.26 ptrgl.s
_MarkingScheme::initializeMarkMap(MM_Environment*)   0.25 MarkingScheme.cpp
.MM_HeapVirtualMemory::getHeapBase()                 0.23 eapVirtualMemory.cpp
```

This output shows that the most-used functions are:

```
MarkingScheme::scanMixedObject(MM_Environment*,J9Object*)
                                       2.67 MarkingScheme.cpp
MarkingScheme::scanClass(MM_Environment*,J9Class*)
                                       2.54 MarkingScheme.cpp
ObjectSlotIterator.GC_ConstantPoolObjectSlotIterator::nextSlot()
                                       1.96 ObjectSlotIterator.cpp
ParallelTask::handleNextWorkUnit(MM_EnvironmentModron*)
                                       1.05 ParallelTask.cpp
```

The values show that the time is being spent during the Mark phase of GC. Because the output also contains references to the Compact and Sweep phases, the GC is probably completing but that it is occurring continuously. You could confirm that likelihood by running with **-verbosegc** enabled.

The same methodology shown above can be used for any case where the majority of the CPU time is shown to be in the Kernel and Shared columns. If, however, the CPU time is classed as being “Other”, a different methodology is required because tprof does not contain a section that correctly details which Java methods are being run.

In the case of CPU time being attributed to “Other”, you can use a Javadump to determine the stack trace for the TIDs shown to be taking the CPU time, and therefore provide an idea of the work that it is doing. Map the value of TID shown in the tprof output to the correct thread in the Javadump by taking the tprof TID, which is stored in decimal, and convert it to hexadecimal. The hexadecimal value is shown as the “native ID” in the Javadump.

For the example above:

```
Process            PID      TID  Total Kernel   User Shared  Other
=======            ===      ===  ===== ======   ==== ======  =====
./java            7018   819317  16.68   5.55   0.00  11.13   0.00
```

This thread is the one using the most CPU; the TID in decimal is `819317`. This value is `C8075` in hexadecimal, which can be seen in the Javadump:

```
3XMTHREADINFO      "main" (TID:0x300E3500, sys_thread_t:0x30010734,
                             state:R, native ID:0x000C8075) prio=5
4XESTACKTRACE          at java/lang/Runtime.gc(Native Method)
4XESTACKTRACE          at java/lang/System.gc(System.java:274)
4XESTACKTRACE          at GCTest.main(GCTest.java:5)
```

These entries show that, in this case, the thread is calling GC, and explains the time spent in the `libj9gc24.so` shared library.

Memory bottlenecks

If the results of vmstat point to a memory bottleneck, you must find out which processes are using large amounts of memory, and which, if any, of these are growing.

Use the **svmon** tool:

```
> svmon -P -t 5
```

This command outputs:

```
-------------------------------------------------------------------------------
     Pid Command        Inuse      Pin     Pgsp  Virtual   64-bit    Mthrd
   38454 java           76454     1404   100413   144805        N        Y
-------------------------------------------------------------------------------
     Pid Command        Inuse      Pin     Pgsp  Virtual   64-bit    Mthrd
   15552 X              14282     1407    17266    19810        N        N
-------------------------------------------------------------------------------
     Pid Command        Inuse      Pin     Pgsp  Virtual   64-bit    Mthrd
   14762 dtwm            3991     1403     5054     7628        N        N
-------------------------------------------------------------------------------
     Pid Command        Inuse      Pin     Pgsp  Virtual   64-bit    Mthrd
   15274 dtsessi         3956     1403     5056     7613        N        N
-------------------------------------------------------------------------------
     Pid Command        Inuse      Pin     Pgsp  Virtual   64-bit    Mthrd
   21166 dtpad           3822     1403     4717     7460        N        N
```

This output shows that the highest memory user is Java, and that it is using 144805 pages of virtual storage (144805 * 4 KB = 565.64 MB). This is not an unreasonable amount of memory for a JVM with a large Java heap - in this case 512 MB.

If the system is memory-constrained with this level of load, the only remedies available are either to obtain more physical memory or to attempt to tune the amount of paging space that is available by using the **vmtune** command to alter the **maxperm** and **minperm** values.

If the Java process continues to increase its memory usage, an eventual memory constraint will be caused by a memory leak.

I/O bottlenecks

This guide does not discuss conditions in which the system is disk-bound or network-bound.

For disk-bound conditions, use filemon to generate more details of which files and disks are in greatest use. For network conditions, use netstat to determine network traffic. A good resource for these kinds of problems is *Accelerating AIX* by Rudy Chukran (Addison Wesley, 1998).

JVM heap sizing

The Java heap size is one of the most important tuning parameters of your JVM. A poorly chosen size can result in significant performance problems as the Garbage Collector has to work harder to stay ahead of utilization.

See How to do heap sizing for information on how to correctly set the size of your heap.

JIT compilation and performance

When deciding whether or not to use JIT compilation, you must make a balance between faster execution and increased processor usage during compilation.

The JIT is another area that can affect the performance of your program. The performance of short-running applications can be improved by using the **-Xquickstart** command-line parameter. The JIT is switched on by default, but you can use **-Xint** to turn it off. You also have considerable flexibility in controlling JIT processing. For more details about the JIT, see The JIT compiler and "JIT and AOT problem determination" on page 159.

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 Using the HPROF Profiler. **-Xrunhprof:help** gives you a list of suboptions that you can use with hprof.

MustGather information for AIX

The information that is most useful at a point of failure depends, in general, on the type of failure that is experienced. These normally have to be actively generated and as such is covered in each of the sections on the relevant failures. However, some data can be obtained passively:

The AIX core file

If the environment is correctly set up to produce full AIX Core files (as detailed in "Setting up and checking your AIX environment" on page 32), a core file is generated when the process receives a terminal signal (that is, SIGSEGV, SIGILL, or SIGABORT). The core file is generated into the current working directory of the process, or at the location pointed to by the label field specified using **-Xdump**.

For complete analysis of the core file, the IBM support team needs:

- The core file
- A copy of the Java executable that was running the process
- Copies of all the libraries that were in use when the process core dumped

When a core file is generated:

1. Run the jextract utility against the core file like this

```
jextract <core file name>
```

   to generate a file called dumpfilename.zip in the current directory. This file is compressed and contains the required files. Running jextract against the core file also allows the subsequent use of the Dump Viewer
2. If the jextract processing fails, use the snapcore utility to collect the same information. For example, `snapcore -d /tmp/savedir core.001 /usr/java5/jre/bin/java` creates an archive (snapcore_pid.pax.Z) in the file /tmp/savedir.

You also have the option of looking directly at the core file by using dbx. However, dbx does not have the advantage of understanding Java frames and the JVM control blocks that the Dump Viewer does. Therefore, you are recommended to use the Dump Viewer in preference to dbx.

The javacore file:

When a javacore file is written, a message (JVMDUMP010I) is written to stderr telling you the name and full path of the javacore file. In addition, a javacore file can be actively generated from a running Java process by sending it a **SIGQUIT (kill -QUIT or Ctrl-\)** command.

The Error Report

The use of **errpt -a** generates a complete detailed report from the system error log. This report can provide a stack trace, which might not have been generated elsewhere. It might also point to the source of the problem where it is otherwise ambiguous.

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.

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. 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, also known as Comm tracing. Every incoming and outgoing GIOP message is 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

Class/Interface	Method	Required permission
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 96.

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 -Xgcpolicy:metronome -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 159).
- 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.

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.

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.

Characters displayed in the console but not in the SDK Components and vice versa (AIX).

Characters that should be displayed in the console are handled by the native operating system. Thus, if the characters are not displayed in the console, in AIX use the `xlfd <physical font name>` command to check whether the system can recognize the character or not.

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 7. Using diagnostic tools

Diagnostics tools are available to help you solve your problems.

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.

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 -Xgcpolicy:metronome -Xdump:heap:none -Xdump:heap+java:events=vmstart+vmstop <class> [args...
```

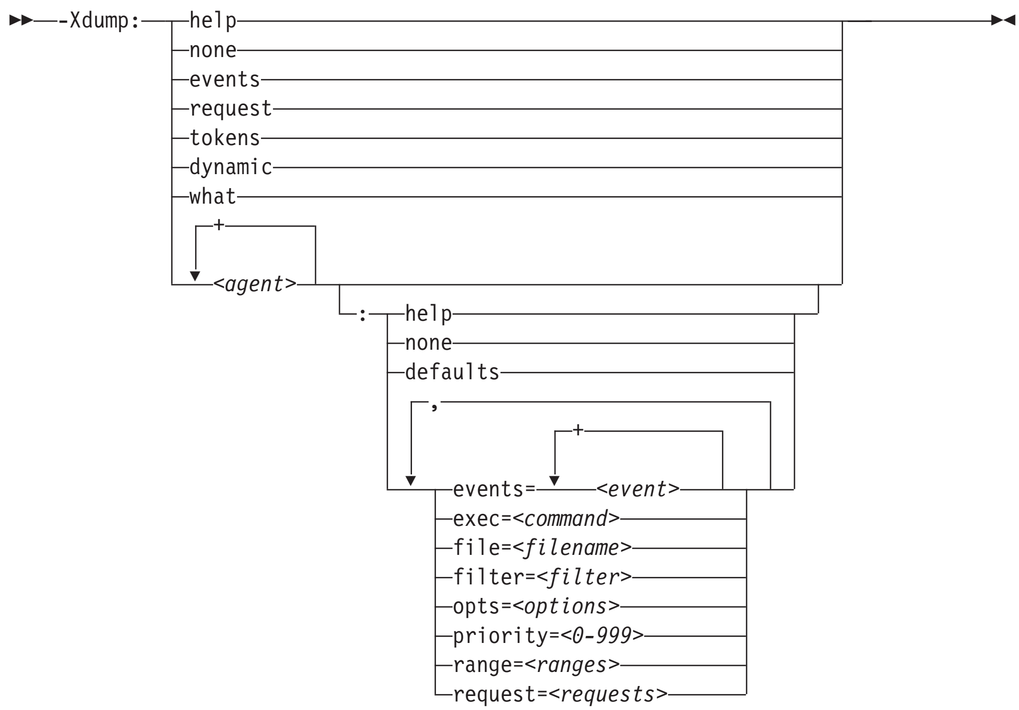
turns off all Heapdumps and create a dump agent that produces a Heapdump and a Javadump when either a vmstart or vmstop event occurs.

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 -Xgcpolicy:metronome "-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 111.
tool	Run command-line program.
java	Write application summary. See "Using Javadump" on page 95.
heap	Capture heap graph. See "Using Heapdump" on page 107.
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 -Xdump:console:events=thrstart,range=1..1 -Xgcpolicy:metronome -version

JVMDUMP006I Processing dump event "thrstart", detail "" - please wait.
---- Console dump --------

Stack Traces of Threads:


ThreadName=Signal Dispatcher(00000001101B3B00)
Status=Running

ThreadName=main(000000011011FA18)
Status=Waiting
Monitor=00000001101A6438 (Thread public flags mutex)
Count=0
Owner=(0000000000000000)
In com/ibm/misc/SystemIntialization.lastChanceHook()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 "".
```

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. In this example, the file name is overridden from the default.

```
java -Xgcpolicy:metronome -Xdump:system:events=vmstop,file=my.dmp

  ::::::::: removed usage info :::::::::

JVMDUMP006I Processing Dump Event "vmstop", detail "#00000000" - Please Wait.
JVMDUMP007I JVM Requesting System Dump using '/home/user/my.dmp'
JVMDUMP010I System Dump written to /home/user/my.dmp
JVMDUMP013I Processed Dump Event "vmstop", detail "#00000000".
```

See "Using system dumps and the dump viewer" on page 111 for more information about analyzing a system dump.

Related information

"Using system dumps and the dump viewer" on page 111
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 91. If you do not specify a tool to use, a platform specific debugger is started.

```
java -Xgcpolicy:metronome -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 90 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 -Xgcpolicy:metronome -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 95 for more information about analyzing a Javadump.

Related information

“Using Javadump” on page 95
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 107 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 -Xgcpolicy:metronome -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 107 for more information about analyzing a Heapdump.

Related information

“Using Heapdump” on page 107
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 -Xgcpolicy:metronome -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 148 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 88 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***
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 85 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 -Xgcpolicy:metronome -Xdump:heap:events=vmstop,file=my.dmp
```

You can use tokens to add context to dump file names. See "Dump agent tokens" on page 91 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[#stackFrame(
```

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.

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 107 for more information.

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 -Xgcpolicy:metronome -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 91

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.

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

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 92. 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 -Xgcpolicy:metronome -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 90.
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" 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 107 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 AIX

AIX produces a core file called `core` in the working directory when a process crashes. The file can be processed with jextract and analyzed with tools such as jdmpview and DTFJ. For the dump to be useful, configure your AIX environment to produce full core dumps. See "Setting up and checking your AIX environment" on page 32 for more details.

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 81) using **–Xdump:java:** on application startup. You can also control Javadumps by the use of environment variables. See "Environment variables and Javadump" on page 106.

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.

Related information

"Javadumps" on page 85
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 81 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 the event 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 6, "Problem determination," on page 31 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 AIX 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.

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 81 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 145

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 199 for more information.

An example tag (`1CIJAVAVERSION`) is shown as follows:

```
1CIJAVAVERSION J2RE 6.0 IBM J9 2.5 AIX ppc64-64 build jvmap64srt60sr2-20090528_36265
```

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: 2009/06/17 at 05:57:17
1TIFILENAME    Javacore filename: /home/user/javacore.20090617.055717.254416.0001.txt
NULL           ------------------------------------------------------------------
0SECTION       GPINFO subcomponent dump routine
NULL           ================================
2XHOSLEVEL     OS Level : AIX 6.1
2XHCPUS        Processors -
3XHCPUARCH       Architecture : ppc64
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 . 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
OTHER	0x00000
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:00000000`, which indicates a crash outside any of the major JVM components. This typically means that the failure occurred when executing Java code; in this case when the Java application calls into the -Xtrace option library `libj9trc25.so`.

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 159.

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:                2009/06/11 at 09:44:39
1TIFILENAME    Javacore filename:    /home/user/javacore.20090611.094432.294936.0003.txt
NULL           ------------------------------------------------------------------------
0SECTION       GPINFO subcomponent dump routine
NULL           ================================
2XHOSLEVEL     OS Level         : AIX 6.1
2XHCPUS        Processors -
3XHCPUARCH       Architecture   : ppc64
3XHNUMCPUS       How Many       : 8
3XHNUMASUP       NUMA is either not supported or has been disabled by user
NULL
1XHEXCPCODE    J9Generic_Signal_Number: 0000000000000004
1XHEXCPCODE    Signal_Number: 000000000000000B
1XHEXCPCODE    Error_Value: 0000000000000000
1XHEXCPCODE    Signal_Code: 0000000000000033
1XHEXCPCODE    Handler1: 09001000A063EEA0
1XHEXCPCODE    Handler2: 09001000A0636508
NULL
1XHEXCPMODULE  Module: /home/user/sdk/jre/lib/ppc64/softrealtime/libj9trc25.so
1XHEXCPMODULE  Module_base_address: 0900000003C07000
NULL
1XHREGISTERS   Registers:
2XHREGISTER      R0: 000000000000000A
2XHREGISTER      R1: 0000000110116C10
2XHREGISTER      R2: 09001000A06546B8
2XHREGISTER      R3: 0000000000000000
2XHREGISTER      R4: 09001000A062A260
2XHREGISTER      R5: 000000000000000A
2XHREGISTER      R6: 09001000A0654030
2XHREGISTER      R7: 0000000000000048
2XHREGISTER      R8: 00000000101771FB
....
2XHREGISTER      FPR30: 0000000000000000
2XHREGISTER      FPR31: 0000000000000000
2XHREGISTER      IAR: 0900000003C0C91C
2XHREGISTER      LR: 0900000003C0CC5C
2XHREGISTER      MSR: A00000000000D032
2XHREGISTER      CTR: 0900000003C0B954
2XHREGISTER      CR: 8400428920000005
2XHREGISTER      FPSCR: 8202400000000000
2XHREGISTER      XER: 2000000582024000
NULL
1XHFLAGS       VM flags:0000000000000000
NULL
NULL           ------------------------------------------------------------------------
0SECTION       ENVINFO subcomponent dump routine
NULL           =================================
1CIJAVAVERSION J2RE 6.0 IBM J9 2.5 AIX ppc64-64 build jvmap64srt60sr2-20090528_36265
1CIVMVERSION   VM build 20090528_036265
1CIJITVERSION  JIT enabled, AOT enabled - r10_20090527_2033
1CIGCVERSION   GC - 20090508_AA
1CIRUNNINGAS   Running as a standalone JVM
1CICMDLINE     sdk/jre/bin/java -Xgcpolicy:metronome -Xtrace:trigger=Method{*.*,segv} -version
1CIJAVAHOMEDIR Java Home Dir:   /home/user/sdk/jre
```

```
1CIJAVADLLDIR  Java DLL Dir:    /home/user/sdk/jre/bin
1CISYSCP       Sys Classpath:   /home/user/sdk/jre/lib/ppc64/softrealtime/jclSC160/vm.jar....
1CIUSERARGS    UserArgs:
2CIUSERARG               -Xjcl:jclscar_25
2CIUSERARG               -Dcom.ibm.oti.vm.bootstrap.library.path=/home/user/sdk/jre/lib/ppc64/softrealtime....
2CIUSERARG               -Dsun.boot.library.path=/home/user/sdk/jre/lib/ppc64/softrealtime....
2CIUSERARG               -Djava.library.path=/home/user/sdk/jre/lib/ppc64/softrealtime....
2CIUSERARG               -Djava.home=/home/user/sdk/jre
2CIUSERARG               -Djava.ext.dirs=/home/user/sdk/jre/lib/ext
2CIUSERARG               -Duser.dir=/home/user
2CIUSERARG               _j2se_j9=1119744 0x09001000A06292A8
2CIUSERARG               -Djava.runtime.version=pap6460srtsr2-20090602_01 (SR2)
2CIUSERARG               -Xdump
2CIUSERARG               -Djava.class.path=.
2CIUSERARG               -Xgcpolicy:metronome
2CIUSERARG               -Xtrace:trigger=Method{*.*,segv}
2CIUSERARG               -Dsun.java.launcher=SUN_STANDARD
2CIUSERARG               _port_library 0x09001000A062A260
2CIUSERARG               _org.apache.harmony.vmi.portlib 0x0000000110135428
```

In the example above, the following lines show where the crash occurred:

```
1XHEXCPMODULE  Module: /home/user/sdk/jre/lib/ppc64/softrealtime/libj9trc25.so
1XHEXCPMODULE  Module_base_address: 0900000003C07000
```

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. It also contains garbage collection history data, described in “Default memory management tracing” on page 128. 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 16 and “Snap traces” on page 86 for more details.

The following example shows some typical output. All the values are output as hexadecimal values.

```
0SECTION MEMINFO subcomponent dump routine
NULL =================================
NULL
1STMEMTYPE Object Memory
NULL region start end size name
1STHEAP 0x00000001109B4090 0x0700000000000000 0x0700000020000000 0x0000000020000000 Default
NULL
1STMEMUSAGE Total memory available: 0000000536870912 (0x0000000020000000)
1STMEMUSAGE Total memory in use: 0000000000262144 (0x0000000000040000)
1STMEMUSAGE Total memory free: 0000000536608768 (0x000000001FFC0000)
NULL
1STSEGTYPE Internal Memory
NULL segment start alloc end type size
1STSEGMENT 0x00000001109B2720 0x0000000111DF3288 0x0000000111DF3288 0x0000000111E432B8 0x01000040 0x0000000000050030
1STSEGMENT 0x00000001109B2670 0x0000000111DA9FC8 0x0000000111DA9FC8 0x0000000111DF3238 0x01000040 0x0000000000049270
1STSEGMENT 0x00000001109B25C0 0x0000000111D66928 0x0000000111D66928 0x0000000111DA9F80 0x01000040 0x0000000000043658
1STSEGMENT 0x00000001109B2510 0x0000000110BFF5E8 0x0000000110BFF5E8 0x0000000110C26718 0x01000040 0x0000000000027130
NULL
1STSEGUSAGE Total memory available: 0000000001194536 (0x0000000000123A28)
1STSEGUSAGE Total memory in use: 0000000000000000 (0x0000000000000000)
1STSEGUSAGE Total memory free: 0000000001194536 (0x0000000000123A28)
```

```
NULL
1STSEGTYPE Class Memory
NULL segment start alloc end type size
1STSEGMENT 0x00000001109B39A0 0x09001000A4EB0BC0 0x09001000A4EB1064 0x09001000A4EB1064 0x04020104 0x00000000000004D4
1STSEGMENT 0x00000001109B38F0 0x09001000A4E09F30 0x09001000A4E0A3A8 0x09001000A4E0A3A8 0x04200104 0x00000000000004A8
1STSEGMENT 0x00000001109B3840 0x0000000111E7CBD0 0x0000000111E84490 0x0000000111E84BD0 0x00010040 0x0000000000008010
1STSEGMENT 0x00000001109B3790 0x0000000111E5CB88 0x0000000111E73AE0 0x0000000111E7CB88 0x00020040 0x0000000000020000
NULL
1STSEGUSAGE Total memory available: 0000000000166284 (0x000000000002898C)
1STSEGUSAGE Total memory in use: 0000000000127284 (0x000000000001F134)
1STSEGUSAGE Total memory free: 0000000000039000 (0x0000000000009858)
NULL
1STSEGTYPE JIT Code Cache
NULL segment start alloc end type size
1STSEGMENT 0x0000000110BDECD0 0x0000000110C27360 0x0000000111427360 0x0000000111427360 0x00000068 0x0000000000800020
NULL
1STSEGUSAGE Total memory available: 0000000008388640 (0x0000000000800020)
1STSEGUSAGE Total memory in use: 0000000008388608 (0x0000000000800000)
1STSEGUSAGE Total memory free: 0000000000000032 (0x0000000000000020)
NULL
1STSEGTYPE JIT Data Cache
NULL segment start alloc end type size
1STSEGMENT 0x0000000110BDF190 0x000000011145EF08 0x000000011146EF68 0x0000000111C5EF08 0x00000048 0x0000000000800000
NULL
1STSEGUSAGE Total memory available: 0000000008388608 (0x0000000000800000)
1STSEGUSAGE Total memory in use: 0000000000065632 (0x0000000000010060)
1STSEGUSAGE Total memory free: 0000000008322976 (0x00000000007EFFA0)
```

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 list of Java threads and stack traces.

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)
```

A Java dump that is produced from a no-heap real-time thread could have some missing information. If 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 AIX on 64-bit POWER 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 3. New thread names in WebSphere Real Time for AIX on 64-bit POWER

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 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.

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.

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 93 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.

Related information

"Heapdumps" on page 86
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 81 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 81 for more information.

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 108 for more details.

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" on page 108. 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 81). 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

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 16.

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 93 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 110. 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."

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 81 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 93 for more information.

Related information

"System dumps" on page 85
System dumps involve dumping the address space and as such are generally very large.

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.

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 dbx, 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_base_address=7F8D0000 Offset_in_DLL=001e506d

Method_being_compiled=org/junit/runner/JUnitCore.runMain([Ljava/lang/String;)Lorg/junit/runner/Resul
```

Dump agents are the primary method for controlling the generation of system dumps. See "Using dump agents" on page 81 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 (-Xgcpolicy:metronome 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 -Xgcpolicy:metronome <core_file> [<zip_file>]
```

or

```
jextract -Xgcpolicy:metronome -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).
```

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.

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 -Xgcpolicy:metronome -zip <zip file>
```

or

```
jdmpview -Xgcpolicy:metronome -core <core file> [-xml <xml file>]
```

The jdmpview-Xgcpolicy:metronome 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 -Xgcpolicy:metronome 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 -Xgcpolicy:metronome -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 -Xgcpolicy:metronome is an interactive, command-line tool to explore the information from a JVM system dump and perform various analysis functions.

cd *<directory_name>*
: Changes the working directory to *<directory_name>*. The working directory is used for log files. Logging is controlled by the **set logging** command. Use the **pwd** command to query the current working directory.

deadlock
: This command detects deadlock situations in the Java application that was running when the system dump was produced. Example output:

```
deadlock loop:
thread: Thread-2 (monitor object: 0x9e32c8) waiting for =>
thread: Thread-3 (monitor object: 0x9e3300) waiting for =>
thread: Thread-2 (monitor object: 0x9e32c8)
```

Threads are identified by their Java thread name, whereas object monitors are identified by the address of the object in the Java heap. You can obtain further information about the threads using the **info thread *** command. You can obtain further information about the monitors using the **x/J <0xaddr>** command.

In this example, the deadlock analysis shows that `Thread-2` is waiting for a lock held by `Thread-3`, which is in turn waiting for a lock held earlier by `Thread-2`.

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 shows the number of matching addresses you specify with *<matches_to_display>*. You can also display the next *<bytes_to_print>* bytes for the last match.

By default, the **find** command searches for the 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 every <*memory_boundary*> bytes. For example, if you specify a <*memory_boundary*> of "4", the command searches for the pattern every 4 bytes.

findnext
: Finds the next instance of the last string passed to **find** or **findptr**. It repeats the previous **find** or **findptr** command, depending on which one was issued last, 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 shows the number of matching addresses you specify with <*matches_to_display*>. You can also display the next <*bytes_to_print*> bytes for the last match.

By default, the **findptr** command searches for the 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 every <*memory_boundary*> bytes. For example, if you specify a <*memory_boundary*> of "4", the command searches for the pattern every 4 bytes.

help [<*command_name*>**]**
: Shows information for a specific command. If you supply no parameters, **help** shows the complete list of supported commands.

info thread [* | <*thread_name*>**]**
: Displays information about Java and native threads. The following information is displayed for 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

If you supply no parameters, the command shows information about the current thread.

info system
: Displays the following information about the system that produced the core dump:

- amount of memory
- operating system
- virtual machine or virtual machines present

info class [<*class_name*>**]**
: Displays the inheritance chain and other data for a given class. If a class name is passed to **info class**, the following information is shown 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

If no parameters are passed to **info class**, the following information is shown:

- the number of instances of each class.
- the total size of all instances of each class.
- the total number of instances of all classes.
- the total size of all objects.

info proc
Displays threads, command-line arguments, environment variables, and shared modules of the 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 shows a list of module sections for each module, 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>*]
If no parameters are passed to this command, the heap names and heap sections are shown.

Using either "*" or a heap name shows 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 with the hexdump command to enable easy scrolling forwards through memory. The previous hexdump command is repeated, starting from the end of the previous one.

-
Displays the previous section of memory in hexdump-like format. This command is used with the hexdump command to enable easy scrolling backwards through memory. The previous hexdump command is repeated, 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 parameter enables 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. The path is 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 takes 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 are appended to the log file. When overwrite is on, the log file is overwritten after the **set logging** command. (Default: off)

redirect [on|off]
Turns redirecting to file on or off, with off being the default. When logging is set to on:

- a value of on for **redirect** sends non-error output only to the log file.
- a value of off for **redirect** sends non-error output to the console and log file.

Redirect must be turned off before 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 =
- The file that is currently being logged to might 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 like 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 *<class_name>* is supplied, all static fields with their values are shown, followed by all objects of that class with their fields and values. If an object address (in hex) is supplied, static fields for that object's class are not shown; 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 hardware architecture this dump file is from. For example, the file might be from a big endian architecture.

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 hardware architecture this dump file is from. For example, the file might be from a big endian architecture.

Note: This command uses the number of items and unit size passed to it by the **x/** command.

x/K *<0xaddr>*
Where the size is defined by the pointer size of the architecture, this parameter shows the value of each section of memory. The output is adjusted for the hardware 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. Although this command can work on an arbitrary section of memory, it is probably more useful on a section of memory that refers to a stack frame. To find the memory section of a thread 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 -Xgcpolicy:metronome-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.

Command	Sub-command	Description
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.
	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 the core file viewing tool; any log files that are currently open are 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.

Trace is a powerful tool to help you diagnose the JVM.

Related concepts

"Troubleshooting the Metronome Garbage Collector" on page 16
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 132 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 132.)

You can find tracepoint information in the following diagnostics data:

- System memory dumps, extracted using jdmpview.
- Snap traces, generated when the JVM encounters a problem or an output file is specified. "Using dump agents" on page 81 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 work the JVM has performed 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. The effect is that the tracepoints 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 probably contains 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 continues running after the logs have been produced. Assertion failures often indicate a serious problem and the JVM might exit with a subsequent error. Even if the JVM does not encounter another error, restart the JVM as soon as possible. Send a service request to IBM, including the standard error output and the `.trc` and `.dmp` files produced.

When an assertion trace point is hit, a message like 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 diagnostic 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 special trace points. They can be enabled or disabled using the standard trace command-line options. See "Controlling the trace" on page 131 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

Using memory buffers for holding trace data is an efficient method of running trace. The reason is that no file I/O is performed until a problem is detected or until the buffer content is intentionally stored in a file.

Buffers are allocated on a per-thread principle. This principle removes contention between threads, and prevents trace data for an individual thread from being mixed in with trace data from 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 the **-Xtrace:buffers=**<*size*> option to control the size of the buffer allocated to each thread.

Note: On some systems, power management affects the timers that trace uses, and might result in misleading information. For reliable timing information, disable power management.

To examine the trace data captured in these memory buffers, you must snap or dump the data, 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 sent to a snap trace file under certain conditions:
- 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 trace file is placed into the current working directory, with a name in the format `Snap`*`nnnn.yyyymmdd.hhmmssth.process`*`.trc`, where *nnnn* is a sequence number reset to 0001 at JVM startup, *yyyymmdd* is the current date, *hhmmssth* is the current time, and *process* is the process identifier. This file is in a binary format, and requires the use of the supplied trace formatter so that you can read it.

You can use the **-Xdump:snap** option to vary the events that cause a snap trace file to be produced.

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 153.

Trace combinations

Most forms of trace can be combined, with the same or different trace data going to different destinations.

The exceptions to this are "in-memory trace" and "trace to a file". These traces are mutually exclusive. When an output file is specified, any trace data that wraps in the "in-memory" case is written to the file, and a new buffer is given to the thread that filled its buffer. If no output file is specified, then when the buffer for a thread is full, the thread wraps the trace data back to the beginning of the 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 132 for the appropriate trace option description.

Specifying trace options

The preferred way to control trace is through trace options that you specify by using the **-Xtrace** option on the launcher command line, or by using 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 file names 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: The requirement to use braces applies only to options specified on the command line. You do not need to use braces for options specified in a properties file.

The syntax for specifying trace options depends on the launcher. Usually, it is:

```
java -Xgcpolicy:metronome -Xtrace:<name>,<another_name>=<value> HelloWorld
```

To switch off all tracepoints, use this option:

```
java -Xgcpolicy:metronome -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:

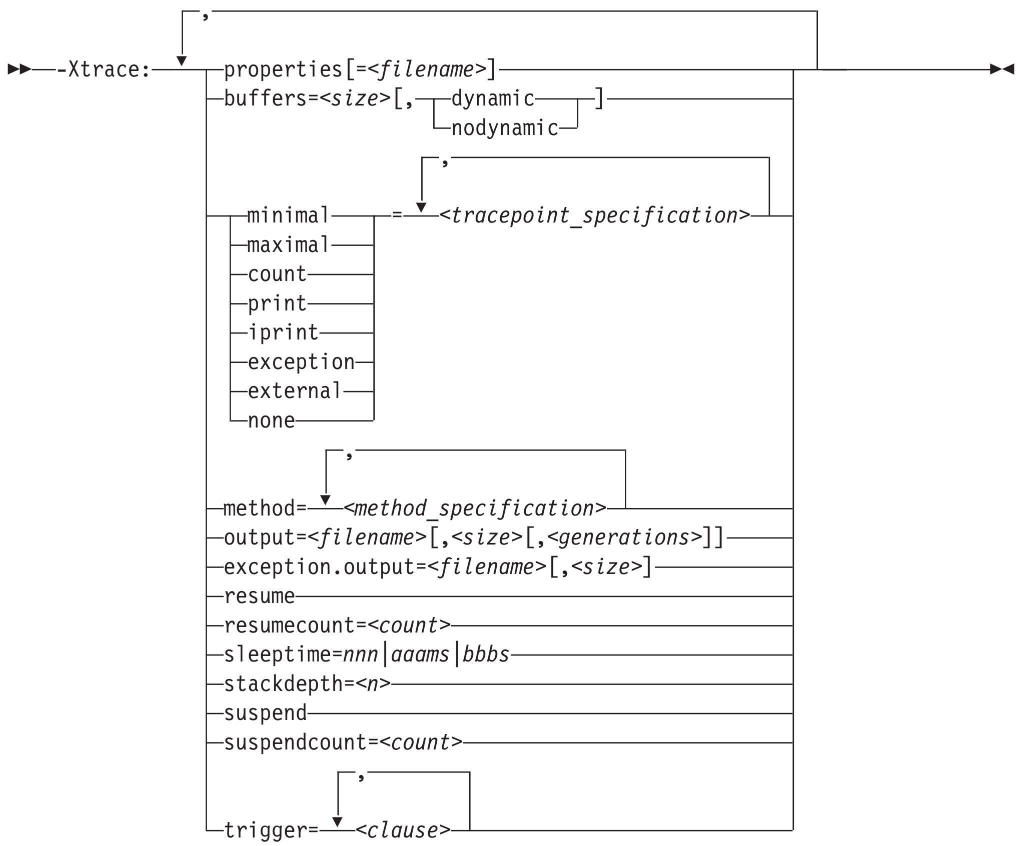
```
java -Xgcpolicy:metronome "-Xtrace:<name>,<another_name>=<value>" HelloWorld
```

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 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 137.

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 128. 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 -Xgcpolicy:metronome
```
- No effect apart from ensuring that the trace engine is loaded (which is the default behavior):
```
java -Xgcpolicy:metronome -Xtrace
```
- Trace engine is not loaded:
```
java -Xgcpolicy:metronome -Xtrace:none
```
- Trace engine is loaded, but no tracepoints are captured:
```
java -Xgcpolicy:metronome -Xtrace:none=all
```
- Default options applied, with the addition of printing for j9vm.209
```
java -Xgcpolicy:metronome -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 -Xgcpolicy:metronome -Xtrace:none -Xtrace:iprint=j9vm.209
```
- Printing for j9vm.209 only:
```
java -Xgcpolicy:metronome -Xtrace:none,iprint=j9vm.209
```
- Default tracing for all components except j9vm, with printing for j9vm.209:
```
java -Xgcpolicy:metronome -Xtrace:none=j9vm,iprint=j9vm.209
```
- Default tracing for all components except j9vm, with printing for j9vm.209
```
java -Xgcpolicy:metronome -Xtrace:none=j9vm -Xtrace:iprint=j9vm.209
```
- No tracing for j9vm (none overrides iprint):
```
java -Xgcpolicy:metronome -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 149.

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:

```
                 ┌ , ──────────────────────────────────────────────────────────┐
                 ↓                                                             │
►►──methods=──{──┴──[!]───┬─*──────────────────────────────┬─.─┬─*──────────────┬─┴──}──────────►◄
                          └─[*][<package>/]<class>[*]──────┘   ├─[*]<method>[*]─┤
                                                               └─[()]───────────┘
```

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 142.

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*> **resumethis** actions before it starts tracing.

Note: You cannot use **resumecount** and **suspendcount** together because they use the same internal counter.

This system property is for use with the **trigger** property. For more information, see "trigger=<*clause*>[,<*clause*>][,<*clause*>]..." on page 145.

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 need 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 a method that matches <*methodspec*>, the specified <*exitAction*> is run. 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 107.

javadump
Produce a Javadump. See "Using Javadump" on page 95.

jstacktrace
Examine 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 examined with the **stackdepth=n** option. See "stackdepth=<n>" on page 144. 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 144.

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 111.

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 81 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{mycom
  ```

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 type:

```
java com.ibm.jvm.format.TraceFormat -Xgcpolicy:metronome <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.

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.

-thread: *<thread id>***[,***<thread id>***]...**
: Filters the output for the given thread IDs only. *thread id* is the ID of the thread, which can be specified in decimal or hex (0x) format. 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:

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 run time. 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 that are requested on the 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 -Xgcpolicy:metronome 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 -Xgcpolicy:metronome -Xtrace:iprint=HelloWorld HelloWorld
```

See "Options that control tracepoint activation" on page 135 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/AttachHand
  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/util
  e0
  10:37:42.246 0x1a1100 mt.6 < java/util/HashMap.createHashedEntry(Ljava/lang/Object;II)Ljava/util
  10:37:42.256 0x1a1100 mt.1 > java/util/HashMap.findNonNullKeyEntry(Ljava/lang/Object;II)Ljava/ut
  d7e0
  10:37:42.266 0x1a1100 mt.7 < java/util/HashMap.findNonNullKeyEntry(Ljava/lang/Object;II)Ljava/ut
  ```

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.

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
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

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 compiled by the AOT compiler start with `+ (AOT cold)`. 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. Optional: If you are diagnosing an AOT problem, run the program a second time with the same options to allow compiled methods to be loaded from the shared data cache. You can also add the **–Xaot:scount=0** option to ensure that AOT-compiled methods stored in the shared data cache will be used when the method is first called. Some AOT compilation failures happen only when AOT-compiled code is loaded from the shared data cache. To help diagnose these problems, use the **–Xaot:scount=0** option to ensure that AOT-compiled methods stored in the shared data cache are used when the method is first called, which might make the problem easier to reproduce. Please note that if you set the **scount** option to 0 it will force AOT code loading and will pause any application thread waiting to execute that method. Thus, this should only be used for diagnostic purposes. More significant pause times can occur with the **–Xaot:scount=0** option.
4. 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 compiled or loaded from the shared data cache, exclude the method from AOT compilation and AOT loading using the **exclude**={<*method*>} parameter:

```
-Xaot:exclude={java/lang/Math.max(II)I}
```

AOT methods are compiled at the “cold” optimization level only. Preventing AOT compilation or AOT loading is the best approach for these methods.

Identifying JIT 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 98

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 starts to compile a method, and when it ends. If the JIT fails on a particular method (that is, it starts compiling, but crashes before it can end), use the **exclude** parameter to exclude it from compilation (refer to “Locating the failing method” on page 161). If excluding the method prevents the crash, you have a workaround that you can use while the service team corrects your problem.

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 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 160 for more information.

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 167 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 `OutOfMemoryErrors` 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 167. 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 167.

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 Diagnostics 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 Diagnostics 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 Diagnostics 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 Diagnostics 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 16. 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 .

Deploying shared classes

You cannot enable class sharing without considering how to deploy it sensibly for your 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 174 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 gives 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 administer operating system permissions. A cache created by one user cannot be opened read-write by other users, but other users can reduce startup time by opening the cache as read-only. Opening a cache as read-only also prevents corruption of the cache. This option can be useful on production systems where one instance of an application corrupting the cache might 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 is disabled for the modified or moved containers for that JVM.

Related information

"Security considerations for the shared class cache" on page 4
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 waste resources that might 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 removed. 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 removing stale caches is that they take up disk space.

Non-persistent caches exist in shared memory and retain system resources that might be used 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.

To perform housekeeping functions successfully, whether automatically or explicitly, you must have the correct operating system permissions. In general, if a user has the permissions to open a cache with read-write access, they also have the permissions to remove it. The only exception is for non-persistent caches on Linux, AIX, z/OS, and i5/OS. These caches can be removed only by the user that created the cache. Caches can only be removed 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 removes a named cache

destroyAll
: This suboption removes 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 removed.

expire=0
: This suboption is the same as **destroyAll**.

expire=10000
: This suboption removes 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 remove 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 corrupt or was created by a different service release. The JVM attempts to remove 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. The cache is removed 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 use optimizations to maintain performance under most circumstances. However, there are configurable factors that can affect shared classes performance.

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.

When a classloader opens and reads a `.jar` file, a lock can be obtained on the file. Shared classes assume that the `.jar` file remains locked and so need not be checked continuously.

`.class` files can be created or deleted from a directory at any time. If you include a directory name in a classpath, shared classes performance can be affected because the directory is constantly checked for classes. The impact on performance might be greater if the directory name is near the beginning of the classpath string. For example, consider a classpath of `/dir1:jar1.jar:jar2.jar:jar3.jar;`. When loading any class from the cache using this classpath, 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 package name of the class. 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 another JVM loads the same class A from myOtherApp2.jar, only one ROMClass exists in the cache. However there are two pieces of metadata that describe the source locations.

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. The extra information that must be added about the classes from the second application is minimal.

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. 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 181. 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 obtain any operating system locks to read the data. This behavior can make cache access slightly faster.

However, if any containers of cached classes are changed or moved on a classpath, then sharing is disabled for all classes on that classpath. There are two reasons why sharing is disabled:

1. The JVM is unable to update the cache with the changes, which might affect other JVMs.
2. The cache code does not continually recheck for updates to containers every time a class is loaded because this activity is too expensive.

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 ends, but all other JVMs are 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 option has a 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 enables any subsequent JVMs attaching to the cache to start faster. AOT data is generated for methods that are 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, turn off caching of AOT code.

To diagnose AOT issues, use the **-Xshareclasses:verboseAOT** command-line option. This option generates messages when AOT code is found or stored in the cache. 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 makes more efficient use of cache space by sharing strings between classes, and ensuring that classes are not duplicated. However, there are also command-line options that 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. **-Xshareclasses:noaot** prevents the JVM from storing any AOT data.

-Xshareclasses:nobootclasspath disables the sharing of classes on the boot classpath, 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 classpaths

When a class is loaded from the shared class cache, the stored classpath and the classloader classpath are compared. The class is returned by the cache only if the classpaths "match". The match need not be exact, but the result should be the same as if the class were loaded from disk.

Matching very long classpaths is initially expensive, but successful and failed matches are remembered. Therefore, loading classes from the cache using very long classpaths 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 classpath must be added to the cache.

For example, if a classpath with 50 entries is grown using addURL(), you might create 50 unique classpaths in the cache. This gradual growth uses more cache space and has the potential to slow down classpath 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 while a single JVM is writing to it.

When multiple JVMs start at the same time and no cache exists, only one JVM succeeds in creating the cache. When created, the other JVMs start to populate the cache with the classes they require. These JVMs might try to populate the cache with the same classes.

Multiple JVMs concurrently loading the same classes are coordinated to a certain extent by the cache itself. This behavior reduces the effect of many JVMs trying to load and store the same class from disk at the same time.

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 the same way as classloaders that load classes from disk. If a classloader is garbage collected, the ROMClasses it has added to the cache 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 by recording all semaphore IDs and all memory IDs, where the corresponding keys begin with MSB in the range 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 classes from the cache that are out of date or “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 is managed in a way that helps ensure that the following challenges are addressed:

- Java archive and compressed files are usually locked by class loaders when they are in use. The files 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 check for Java archive and compressed file updates.
- `.class` files that are not in a `.jar` file 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 from directories found 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. This structure helps ensure that when checking for updates, the correct directories are searched.

Class files contained in jars and compressed files, and class files stored as `.class` files on the file system, are accessed and used in different ways. The result is that the cache treats them 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. As a direct consequence of updating the class data, AOT data is automatically updated.

Storing classes

When a classpath is stored in the cache, the Java archive and compressed files are time stamped. 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 subsequent updates to a file cause an update to the directory holding the file.

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 an attempt is made to add a ROMClass to the cache from a directory, but the ROMClass does not exist as a `.class` file, it is not stored in the cache.

Time stamps can also be used to determine whether a ROMClass being added is a duplicate of one that exists in the cache.

If a classpath entry is updated on the file system, the entry becomes out of sync with the corresponding classpath time stamp in the cache. The classpath is added to the cache again, and all entries time stamped again. When a ROMClass is added to the cache, the cache is searched for entries from the classpath that applies to the caller. Any potential classpath matches are also time stamp-checked. This check ensures that the matches are up-to-date before the classpath is returned.

Finding classes

When the JVM finds a class in the cache, it must 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 calling 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 might mean that another version of the class being returned had already been added earlier in the classpath.

Additionally, any classpath entries that are directories might contain `.class` files that “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 by using the classpath entry, the package names, and the class name, and then looking for the class file. 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 an individual `.class` file is updated, only the class or classes stored from that `.class` file are marked “stale”.

When a Java archive or compressed file classpath entry is updated, all of the classes in the cache that could have been affected by that update are marked stale. This action is taken because the cache does not know the contents of individual jars and compressed files.

For example, in the following class paths where **c** has become stale:

a;b;c;d
: **c** might now contain new versions of classes in **d**. Therefore, classes in both **c** and **d** are all stale.

c;d;a
: **c** might now contain new versions of classes in **d** or **a**, or both. 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. Making a single update to one jar file might cause many classes in the cache to be marked stale. To avoid massive duplication as classes are updated, stale classes can be marked as not stale, or "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 not have updated. When a class loader stores a class, and in doing so effectively "updates" a stale class, you can "redeem" the stale class if you can prove that it has not in fact changed.

For example, assume that class **X** is stored in a cache after obtaining it from location **c**, where **c** is part of the classpath `a;b;c;d`. Suppose **a** is updated. The update means that **a** might now contain a new version of class **X**. For this example, assume **a** does not contain a new version of class **X**. The update marks all classes loaded from **b**, **c**, and **d** as stale. Next, another JVM must load class **X**. The JVM asks the cache for class **X**, but it is stale, so the cache does not return the class. Instead, the class loader fetches class **X** from disk and stores it in the cache, 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 179) 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" on page 183 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. The filter is 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 best 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)
```

You must return true when you implement these functions so that a class can be found or stored in the cache. Use the supplied parameters as required. Make sure that you implement functions that do not take long to run 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 that 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 run time to show classes being stored and found in the cache.

VerboseIO output provides information about the I/O activity occurring with the cache, with basic information about find and store calls. You enable verboseIO output by using the verboseIO suboption of -Xshareclasses. With a cold cache, you see trace like this example

```
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. The bootstrap loader has an ID of 0. In the example trace, classloader 17 follows the classloader hierarchy by asking its parents for the class. Each parent asks the shared cache for the class. Because the class does not exist in the cache, all the find calls fail, so the class is stored by classloader 17.

After the class is stored, you see the following output for subsequent calls:

```
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, because parents ask the cache for the class first. This time, the find call succeeds. With other classloading frameworks, such as OSGi, the parent delegation rules are different. In such cases, the output might be different.

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 181.

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 prints summary information about the specified cache to the standard error output.

The printStats utility is a suboption of **-Xshareclasses**. You can specify a cache name using the **name=**<*name*> parameter. printStats is a cache utility, so the JVM reports the information about the specified cache and then exits.

The following output shows example results after running the printStats utility:

```
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 used to store the classes. For more information, see "Using the Java Helper API" on page 181.

 # AOT methods
: Optionally, ROMClass methods can be compiled and the AOT code stored in the cache. The `# AOT methods` 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. Classes stored using a SharedClassURLHelper are stored with a URL. Classes stored using a SharedClassTokenHelper are stored with a Token. Most classloaders, including the bootstrap and application classloaders, use a SharedClassURLClasspathHelper. The result is that 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, such as when a `.jar` file is rebuilt, a new Classpath is added to the cache. Additionally, if “partitions” or “modification contexts” are used, they are associated with the Classpath, URL, or Token. A Classpath, URL, or Token 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 179.

% 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, providing as much diagnostic information as possible. 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 about the cache specified or the default cache and then exits.

Each JVM that connects to the cache receives a unique ID. 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. The class path 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`. The output also indicates the class path against which the class is stored, and from which index in that class path the class was loaded. In the example, 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`

URLs and tokens are displayed in the same format as class paths. A URL is effectively the same as a class path, but with only one entry. A Token is in a similar format, but it is a meaningless String passed to the Java Helper API.

`ZipCache`

```
1: 0x042FE07C ZIPCACHE: luni-kernel.jar_347075_1272300300_1 Address: 0x042FE094 Size: 7898
1: 0x042FA878 ZIPCACHE: luni.jar_598904_1272300546_1 Address: 0x042FA890 Size: 14195
1: 0x042F71F8 ZIPCACHE: nio.jar_405359_1272300546_1 Address: 0x042F7210 Size: 13808
1: 0x042F6D58 ZIPCACHE: annotation.jar_13417_1272300554_1 Address: 0x042F6D70 Size: 1023
```

The first line in the output indicates that JVM 1 stored a zip entry cache called `luni-kernel.jar_347075_1272300300_1` in the shared cache. The metadata for the zip entry cache is stored at address `0x042FE07C`. The data is written to the address `0x042FE094`, and is 7898 bytes in size. Storing zip entry caches for bootstrap jar files is controlled by the **-Xzero:sharebootzip** sub option, which is enabled by default. The full **-Xzero** option is not enabled by default.

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 126 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 179, 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 176.

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 be sourced 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 183)

The class has been retransformed by JVMTI and cacheRetransformed has not been specified
: As described in "Dealing with runtime bytecode modification" on page 176, 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 178.

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 must 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 might have many potential causes, and it is difficult to provide detailed message information 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. 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. A default cache name is created that includes the username to prevent clashes if different users try to share the same default cache. All shared classes users must also 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. 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 occurs. The failure is because user B does not have permissions to access "myCache". Caches can be removed 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 disk. The main reasons for failing to create the file are insufficient disk space and incorrect file permissions.

2) Acquiring file locks
Concurrent access to persistent caches is controlled using operating system file-locking. File locks cannot be obtained if you try to use a cache that is located on a remote networked file system. For example, an NFS or SMB mount. This option 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. You cannot memory-map the cache file to a remote networked file system, such as an NFS or SMB mount. This option is not supported. Alternatively, memory-mapping might fail if there is insufficient system memory.

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 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. Any remaining shared memory areas that are shared class caches are removed. Follow one of these steps to reset the system and force the JVM to re-create the `javasharedresources` directory:
   - On Linux, AIX, or z/OS, using root authority, start the JVM with **-Xshareclasses**.
   - On i5/OS, using a user that has *ALLOBJ authority, start the JVM with **-Xshareclasses**.

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 178) 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", extensionEventCo

   /* 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
            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 81.

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* data_
```

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 81. 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 84. 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 132.

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 86.

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 86.

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 112. 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.

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 AIX on 64-bit POWER.

Figure 3 on page 203 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 112). 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:

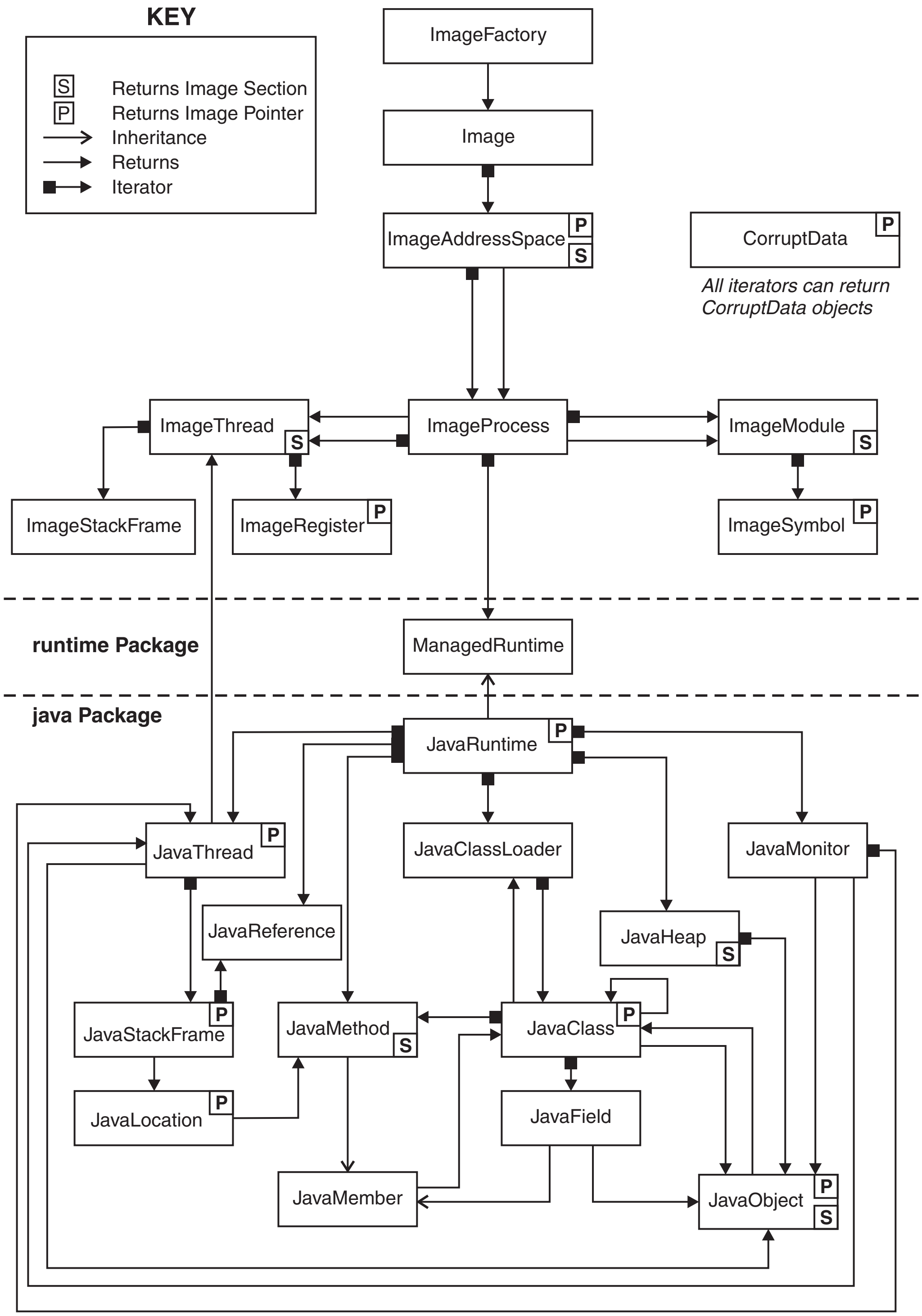


Figure 3. DTFJ interface diagram

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 200.

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 210.
- 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 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 209 for more information.
- Open a log file from disk by canceling the wizard. See "Opening files from disk" on page 218 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 218 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 210 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 6 SR1	No agent included	Profiling, Garbage collection, locking, classes, environment, memory	No	`-agentlib:healthcenter -Xtrace:output=perfmon.%p.out`

Java version	Function with default agent	Function with updated agent	Suitable for production use	Command-line options to enable agent
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 210
2. Start the Java application with the agent enabled. See "Starting a Java application with the Health Center agent enabled" on page 211 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 212 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 216.

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 210 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 207 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 MyMainClas
   ```

   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 213.

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 207 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.agent
```

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 237.

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 210 and "Starting a Java application with the Health Center agent enabled" on page 211 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 237.

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 210 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 210 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 212.

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 210 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 212.
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 210 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 212.
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 207).

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 242.

Related concepts

"Cropping data" on page 242
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:

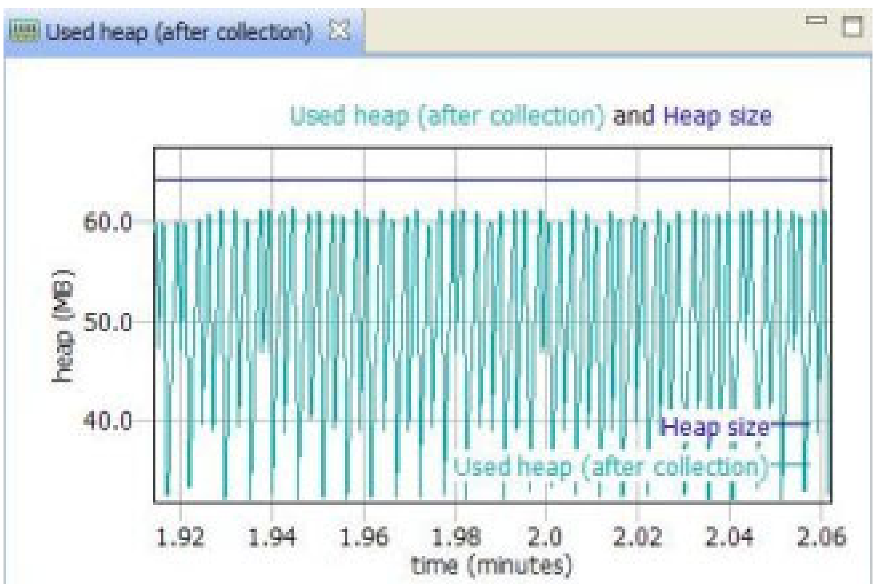
- A graph of heap usage.
- A graph of pause times.
- A summary table of important GC metrics.
- Tuning recommendations.

- 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 242.

Related concepts

“Cropping data” on page 242
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 243.

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 4. 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 5. 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 239 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.

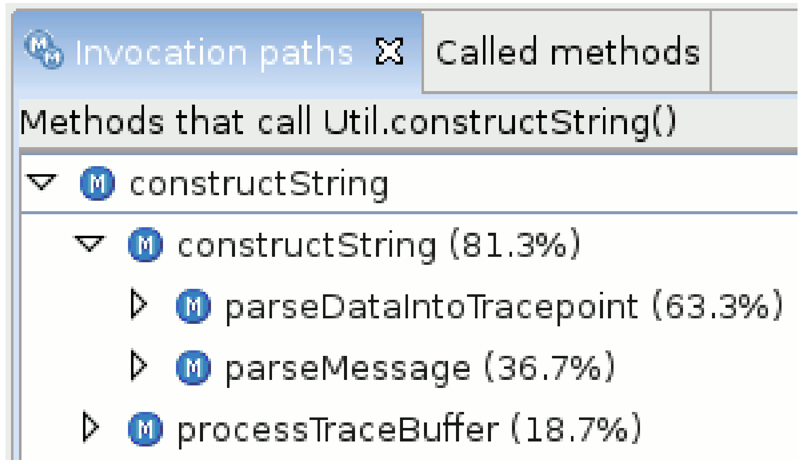
Filter the contents of the method profile table using the text box above the table. See the filtering help topic for more information.

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.



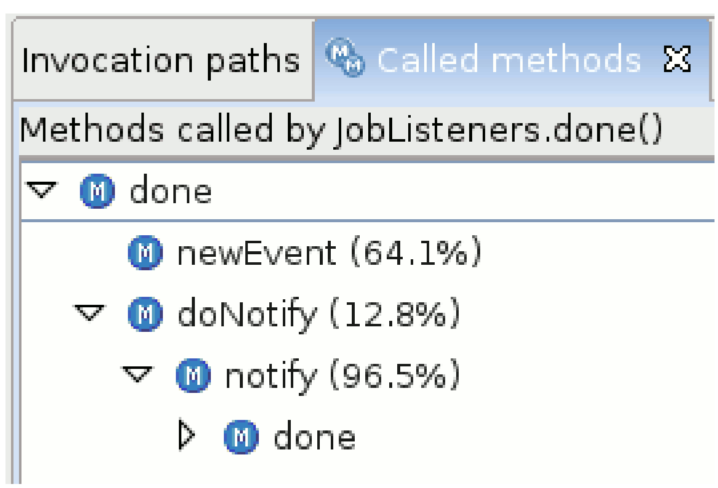
In this case, you have two strategies for optimization. The first is to make the Util.constructString() method more efficient. The second is to reduce how often it is called. Reducing how often processTraceBuffer() calls constructString() makes less difference than halving how often constructString() 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?

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 236.

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 236.

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 236.

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 6 on page 236.

Table 6. 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 234.

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 210 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 211 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 217.

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 207 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

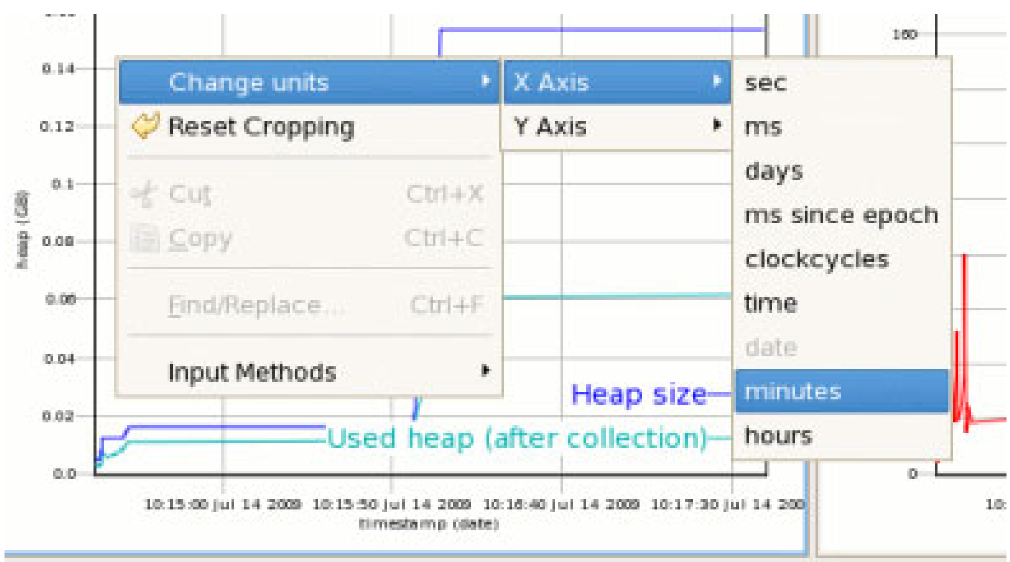
You can reset the graph so that data is no longer cropped in two ways:
- Right-click on the graph. Select **Reset Cropping**.
- Double-click anywhere on the graph.

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 217.

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 218.

Chapter 8. Reference

This set of topics lists the options and class libraries that can be used with WebSphere Real Time for AIX on 64-bit POWER

Real Time specific options

There are a number of command line options that are specific to WebSphere Real Time for AIX on 64-bit POWER.

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>`

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 16.

-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/ppc64/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.

-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 AIX on 64-bit POWER, standard non-real-time mode, and with both Metronome Garbage Collector and WebSphere Real Time for AIX on 64-bit POWER.

Metronome Garbage Collector options

The definitions of the Metronome Garbage Collector options.

-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 -Xgcpolicy:metronome -Xmx30m -Xgc:targetUtilization=75 Test
```

-Xgc:threads=*N*
: Specifies the number of GC threads to run. The default is the number of processor cores available to the process. The maximum value you can specify is the number of processors available to the operating system.

-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.

Related concepts

Chapter 3, "Using the Metronome Garbage Collector," on page 15
Metronome Garbage Collector replaces the standard Garbage Collector in WebSphere Real Time for AIX on 64-bit POWER.

"Introduction to the Metronome Garbage Collector" on page 15
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 16
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.

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.

-Xcompressedrefs
: Uses 32-bit values for references.

-Xnocompressedrefs
: Uses 64-bit values for references. From WebSphere Real Time for AIX on 64-bit POWER V2 SR3, the 64-bit JVM uses compressed references. If you use this option the JVM will not start.

-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.

-Xlp<*size*>
: Requests the JVM to allocate the Java heap (the heap from which Java objects are allocated) with 16 MB large pages, if a size is not specified. If large pages are not available, the Java heap is allocated with AIX's standard 4 KB pages. AIX requires special configuration to enable large pages. For more information on configuring AIX support for large pages, see http://www.ibm.com/servers/aix/whitepapers/large_page.html. The

SDK uses shmget() with the SHM_LGPG and SHM_PIN flags to allocate large pages. By default, large pages are not used.

The options available for <*size*> are:

- -Xlp4K
- -Xlp64K (AIX v5.3 and later)
- -Xlp16M (AIX v5.2 and later)
- -Xlp16G (AIX v5.3 and later)

-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.The default is 64 MB.

-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.

-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.The default is 400® KB.

-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.

-Xrealtime
: Runs the JVM in a real-time mode. In particular, it will run with -Xgcpolicy:metronome

-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.

-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. The default is 1024 KB.

-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 AIX on 64-bit POWER 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
Strict conformancy checks	Disabled	c
Quickstart	Disabled	c
Remote debug info server	Disabled	c
Reduced signalling	Disabled	c

JVM setting	Default	Setting affected by
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	N/A	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 64-bit. Use: **-Xiss<size>**	2 KB	c
Maximum stack size for Java Threads 64-bit. Use: **-Xss<size>**	256 KB	c
Stack size for OS Threads 64-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>**	The number of processor cores available to the process.	c

Note: "available memory" is the smallest of real (physical) memory and the **RLIMIT_AS** value.

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

Numerics

A

B

C

D

E

F

P

R

S

T

U

V

W

Printed in USA