IBM SDK, Java Technology Edition, Version 6

IBM SDK, Java Technology Edition, Version 6, Release 0, Modification 1 Supplement

IBM SDK, Java Technology Edition, Version 6

IBM SDK, Java Technology Edition, Version 6, Release 0, Modification 1 Supplement

Note
Before you use this information and the product it supports, read the information in "Notices" on page 177.

Contents

Chapter 1. Overview

This guide contains supplementary information for IBM® SDK, Java™ Technology Edition, Version 6.

This documentation provides information about additional or changed capabilities for IBM SDK, Java Technology Edition, Version 6 when the product includes the IBM J9 2.6 virtual machine. The information provided here is supplementary to the documentation for IBM SDK, Java Technology Edition, Version 6, which is also contained in this information center.

To find out which IBM J9 virtual machine (VM) you are using with the IBM SDK or runtime environment for Java Version 6, type the following command on the command line:

```
java -version
```

The output includes the build level for the IBM J9 VM. In this example, `build 2.6` indicates the IBM J9 2.6 virtual machine:

```
IBM J9 VM (build 2.6, JRE 1.6.0 ...
```

The information in this guide applies to IBM SDK for z/OS®, Java Technology Edition, Version 6, Release 0, Modification 1 (product numbers 5655-R31 and 5655-R32), and to any other IBM products that include IBM SDK, Java Technology Edition, Version 6 with an IBM J9 2.6 virtual machine, such as:

- WebSphere® Application Server V8.0
- WebSphere Application Server V8.5

For late breaking information that is not in this guide, see IBM SDK Java Technology Edition V6 (J9 VM2.6): Current news.

To determine the service refresh or fix pack level of an installed version, check the second line of the output from the **`java -version`** command. The service refresh (SR), fix pack (FP) and APAR number is appended to the build string. For example: `Java(TM) SE Runtime Environment (build pmz3160_26sr8fp2ifix-20141114_01(SR8 FP2+IV66608+IV66375+IX90155+IV66944))`

Any new modifications made to this user guide are indicated by vertical bars to the left of the changes.

What's new

Read about the features and functions available for this release of IBM SDK, Java Technology Edition, Version 6.

The initial release of IBM SDK, Java Technology Edition, Version 6 (J9 VM 2.6) provides many new features and capabilities, when compared to IBM SDK, Java Technology Edition, Version 6.

Service refreshes provide essential maintenance, including IBM fixes, Oracle Critical Patch Updates (CPUs), and Oracle Synchronized Security Releases (SSRs). Follow these links for more detailed information:

- IBM fixes

- Oracle CPUs and SSRs

In addition to essential maintenance, IBM regularly provides serviceability improvements and performance enhancements to the code base. The following topics capture new capabilities and changes to default behavior. For changes to security providers and utilities, see Chapter 10, "Security," on page 51.

First release

Read about the features and functions available with the initial release of IBM SDK, Java Technology Edition, Version 6 (J9 VM 2.6).

JVM optimizations

New optimizations for Java monitors are available, that are expected to improve CPU efficiency. New locking optimizations are also implemented that are expected to reduce memory usage and improve performance. If you experience performance problems that you suspect are connected to this release, see "Application performance issues" on page 59.

Java Attach API

Connections to virtual machines through the Java Attach API have a new default state. For more information, see Chapter 7, "Developing applications," on page 43.

Garbage Collector policy changes

There is a new garbage collection policy available that is intended for environments where heap sizes are greater than 4 GB. This policy is called the Balanced Garbage Collection policy, and uses a hybrid approach to garbage collection by targeting areas of the heap with the best return on investment. The policy tries to avoid global collections by matching allocation and survival rates. The policy uses mark, sweep, compact and generational style garbage collection. For more information about this policy, see "Balanced Garbage Collection policy" on page 21.

Other policy changes include changes to the default garbage collection policy, and the behavior of specific policies and specific policy options. For more information about these changes, see "Garbage collection policy options" on page 47.

Verbose garbage collection logging

Verbose garbage collection logging has been redesigned. The output from logging is significantly improved, showing data that is specific to the garbage collection policy in force. These changes improve problem diagnosis for garbage collection issues. For more information about verbose logging, see “Verbose garbage collection logging” on page 111.

JIT compilation

The JIT compiler can use more than one thread to convert method bytecodes into native code, dynamically. To learn more about this feature, see “Using more than one JIT compilation thread” on page 48.

Shared classes .zip entry caches

The JVM stores .zip entry caches for bootstrap jar files into the shared cache. A .zip entry cache is a map of names to file positions used to quickly find entries in the .zip file. Storing .zip entry caches is enabled by default, or you can choose to disable .zip entry caching. See “-Xzero” on page 151 for more information.

Shared classes JIT data

You can now store JIT data in the shared class cache, which enables subsequent JVMs attaching to the cache to either start faster, run faster, or both. For more information about improving performance with this option, see “Cache performance” on page 102.

Shared class debug area

A portion of the shared class cache is reserved for storing data associated with JVM debugging. By storing these attributes in a separate region, the operating system can decide whether to keep the region in memory or on disk, depending on whether debugging is taking place. For more information about tuning the class debug area, see “Cache performance” on page 102.

Troubleshooting problems with shared class caches

A new dump event is available that triggers a dump when the JVM finds that the shared class cache is corrupt. This event is added for the system, java, and snap dump agents. For more information about the corrupt cache event, and the default dump agents, see “Using dump agents” on page 63.

Shared Cache Utility APIs

There are new Java Helper APIs available that can be used to obtain information about shared class caches. For more information see “Utility APIs” on page 105.

New IBM JVMTI extensions are included, that can search for shared class caches, and remove a shared class cache. For more information, see “Using the JVMTI” on page 126.

Diagnosing problems with native memory

Information about native memory usage is now provided by a Javadump. For further information, including example output, see “Native memory (NATIVEMEMINFO)” on page 72.

The Diagnostic Tool Framework for Java (DTFJ) interface has also been modified, and can be used to obtain native memory information from a system dump or Javadump. See “Using the DTFJ interface” on page 134.

In addition, you can query native memory usage by using a new IBM JVMTI extension. The GetMemoryCategories() API returns the runtime environment native memory use by memory category. For further information about the IBM JVMTI extension, see “Querying runtime environment native memory categories” on page 126.

JVM message logging

All vital and error messages are now logged by default. However, you can control the messages that are recorded by the JVM using a command-line option. You can also query and modify the message setting by using new IBM JVMTI extensions. For more information about message logging, see “JVM messages” on page 59.

Service refresh 1

This service refresh provides tuning options, and several serviceability improvements.

Compressed references tuning option for z/OS

A new command-line option is available to override the allocation strategy used by the 64-bit JVM when running with compressed references enabled. This option prevents the JVM pre-allocating an area of virtual memory, leaving the operating system to handle the allocation strategy. For more information, see “-XXnosuballoc32bitmem (z/OS)” on page 153.

Balanced Garbage Collection policy change

From service refresh 1, you no longer need to use compressed references with the Balanced Garbage Collection policy.

Subscribing to verbose garbage collection logging with JVMTI extensions

New IBM JVMTI extensions are available to turn on, and turn off, verbose garbage collection logging at run time. For more information, see "Subscribing to verbose garbage collection logging" on page 132.

Reverting to earlier verbose garbage collection logging

A new command-line option is available to revert to the verbose garbage collection logging format available in earlier releases of the J9 VM. See the **-Xgc:verboseFormat** option in "Garbage collection policy options" on page 47.

Improved Java heap shrinkage

New command-line options are available to control the rate at which the Java heap is contracted during garbage collection cycles. You can specify the minimum or maximum percentage of the Java heap that can be contracted at any given time. For more information, see "-Xgc" on page 164.

Changes to locale translation files

From service refresh 1, changes are made to the locale translation files to make them consistent with Oracle JDK 6. To understand the differences in detail, see http://www.ibm.com/support/docview.wss?uid=swg21568667. A system property is available to revert back to the older locale translation files. See "-Dcom.ibm.UseCLDR16" on page 138.

Processing system dumps

To analyze system dumps from Windows and z/OS systems, you no longer need to run the **jextract** utility to process the dump.

For Linux and AIX® platforms, copies of executable files and libraries are required along with the system dump. You must still run the **jextract** utility or the Diagnostics Collector to collect these files. For more information, see "Processing system dumps" on page 91.

System dumps in out-of-memory conditions

A system dump is now generated, in addition to a Heapdump and a Javadump, when an OutOfMemoryError exception occurs in the JVM. The JVM adds a new default dump agent to enable this functionality, see "Default dump agents" on page 67. If you want to disable this new functionality, remove the new dump agent. For more information, see "Removing dump agents" on page 68.

Working with system dumps containing multiple JVMs

For z/OS, service refresh 1 includes an enhanced dump viewer to help you analyze system dumps. For more information, see "Working with dumps containing multiple JVMs" on page 99.

Using the dump viewer in batch mode

For long running or routine jobs, the **jdmpview** command can now be used in batch mode. For more information, see “Using the dump viewer in batch mode” on page 92.

Removing dump agents by event type

You can selectively remove dump agents, by event type, with the **-Xdump** option. This capability allows you to control the contents of a dump, which can simplify problem diagnosis. For more information, see “Removing dump agents” on page 68.

Default assertion tracing during JVM startup

Internal JVM assert trace points are now enabled during JVM startup. For more information, see “Default tracing” on page 100.

Service refresh 2

There are changes to the default heap size on Windows, and several diagnostic improvements to assist with troubleshooting.

Default Java heap size on Windows

From service refresh 2, there is a change in the criteria for setting the default heap size for the JVM. If you do not specify the maximum Java heap size with the **-Xmx** option, the value chosen is half the available memory. The minimum value is 16 MB, and the maximum value is 512 MB. For historical reasons, earlier releases set the heap size based on physical memory size, with a maximum of 2 GB. However, in some situations this criteria results in an inappropriately large heap size, which can lead to out of memory errors. For more information about default settings, see “Default settings for the JVM” on page 171. To understand more about choosing your heap size, see How to do heap sizing.

Using the JVMTI ClassFileLoadHook with cached classes

Historically, the JVMTI ClassFileLoadHook or java.lang.instrument agents do not work optimally with the shared classes cache. Classes cannot be loaded directly from the shared cache unless using a modification context. Even in this case, the classes loaded from the shared cache cannot be modified. The **-Xshareclasses:enableBCI** suboption improves startup performance without using a modification context, when using JVMTI class modification. This suboption allows classes loaded from the shared cache to be modified using a JVMTI ClassFileLoadHook, or a java.lang.instrument agent. The suboption also prevents the caching of modified classes in the shared classes cache, while reserving an area in the cache to store original class byte data for the JVMTI callback. Storing the original class byte data in a separate region allows the operating system to decide

whether to keep the region in memory or on disk, depending on whether the data is being used. You can specify the size of this region, known as the Raw Class Data Area, using the **-Xshareclasses:rcdSize** suboption.

For more information about this capability, see "Using the JVMTI ClassFileLoadHook with cached classes" on page 104. For more information about the **-Xshareclasses** suboptions **enableBCI** and **rcdSize**, see "-Xshareclasses" on page 155.

Using the dump viewer with compressed files

You can now specify the **-notemp** option to prevent the **jdmpview** tool from extracting compressed files before processing them. When you specify a compressed file, the tool detects and shows all core, Java core, and PHD files within the compressed file. Because of this behavior, more than one context might be displayed when you start **jdmpview**. For more information, see "Support for compressed files" on page 90.

Diagnosing problems with locks

The LOCKS section of the Java dump output now contains information about locks that inherit from the java.util.concurrent.locks.AbstractOwnableSynchronizer class.

The THREADS section of the Java dump output now contains information about locks. For more information, see "Understanding Java and native thread details" on page 78.

New dump agent trigger

Dump agents are now triggered if an excessive amount of time is being spent in the garbage collector. The event name for this trigger is **excessivegc**. For more information, see "Dump events" on page 66.

Determining the Linux kernel `sched_compat_yield` setting in force

The ENVINFO section of a javacore contains additional information about the **sched_compat_yield** Linux kernel setting in force when the JVM was started. For more information about the ENVINFO javacore output, see "TITLE, GPINFO, and ENVINFO sections" on page 70.

Receiving OutofMemoryError exceptions

From service refresh 2, an error message is generated when there is an OutofMemoryError condition on the Java heap.

Service refresh 3

There are improvements to hashing algorithms, which can change the iteration order of items returned from hash maps. In addition, further information is provided in a Java dump file to help diagnose problems with direct byte buffers.

- "Improved hashing algorithms" on page 8
- "Diagnosing problems when using Direct Byte Buffers" on page 8

Improved hashing algorithms

An improved hashing algorithm is available for string keys stored in hashed data structures. You can adjust the threshold that invokes the algorithm with the system property, **jdk.map.althashing.threshold**. This algorithm can change the iteration order of items returned from hashed maps.

An enhanced hashing algorithm is also used for javax.xml.namespace.QName.hashCode(). This algorithm can change the iteration order of items returned from hashed maps. You can control the use of this algorithm with the system property, **-Djavax.xml.namespace.QName.useCompatibleHashCodeAlgorithm=1.0**.

For more information about these system properties, see the Java Diagnostics Guide 6.

Diagnosing problems when using Direct Byte Buffers

The JVM contains a new memory category for Direct Byte Buffers. You can find information about the use of this memory category in the NATIVEMEMINFO section of a Javadump. For more information, see "Native memory (NATIVEMEMINFO)" on page 72.

Service refresh 4

This service refresh includes support for 1M pageable large pages, and turns off Java Attach API support by default on z/OS systems. Further serviceability improvements are also available.

Symbol resolution on Linux

By default, the JVM delays symbol resolution for each function in a user native library, until the function is called. Use the **-XX:-LazySymbolResolution** option to force the JVM to immediately resolve symbols for all functions in a user native library when the library is loaded. For more information, see "-XX:[+|-]LazySymbolResolution (Linux only)" on page 153.

Support for 1M pageable large pages

The JVM now includes support for 1M pageable large pages. You can use the **-Xlp** command-line option to instruct the JVM to allocate the Java object heap or the JIT code cache with 1M pageable large pages.

The use of 1M pageable large pages for the Java object heap provides similar runtime performance benefits to the use of 1M nonpageable pages. In addition, using 1M pageable pages provides options for managing memory that can improve system availability and responsiveness.

When 1M pageable large pages are used for the JIT code cache, the runtime performance of some Java applications can be improved.

To take advantage of 1M pageable large pages, the following minimum prerequisites apply: IBM zEnterprise® EC12 with the Flash Express® feature (#0402), z/OS V1.13 with PTFs, APAR OA41307, and the z/OS V1.13 Remote Storage Manager Enablement Offering web deliverable.

For more information, see "-Xlp" on page 146.

IBM z/OS Language Environment®

JVM signal handlers for SIGSEGV, SIGILL, SIGBUS, SIGFPE, SIGTRAP, and for SIGABRT by default terminate the process by using exit(). If you are using the IBM z/OS Language Environment, the Language Environment is not aware that the JVM ended abnormally. Use the **-Xsignal:posixSignalHandler=cooperativeShutdown** option to control how the signal handlers end. For more information, see the Java Diagnostics Guide 6.

Java Attach API support is disabled by default on z/OS

To enhance security on z/OS, support for the Java Attach API is now disabled by default. For more information, see Chapter 7, "Developing applications," on page 43.

Default locking behavior is optimized for the Completely Fair Scheduler (CFS) on Linux

The default locking behavior on Linux systems that are using the CFS in the default mode (`sched_compat_yield=0`) is now optimized to improve performance for most applications. However, if your application uses the Thread.yield() method extensively, you might see a performance decrease in cases where yielding is not beneficial. If you see a performance decrease after upgrading the IBM SDK, you can test whether the new optimizations are negatively affecting your application by reverting to the behavior of earlier versions. To use the earlier behavior, specify the following command-line option:

```
-Xthr:noCfsYield
```

For more information, see "-Xthr" on page 151.

Disabling hardware prefetch on AIX

A new command-line option, **-XXsetHWPrefetch:none**, is available for disabling hardware prefetch on AIX operating systems. This option might help to improve performance for your Java applications. For more information, see "-XXsetHWPrefetch:[none|os-default] (AIX only)" on page 154.

Configuring the initial maximum Java heap size

The **-Xsoftmx** option is now available on Linux, Windows, and z/OS, in addition to AIX. A soft limit for the maximum heap size can be set by using the

com.ibm.lang.management API. The Garbage Collector attempts to respect the new limit, shrinking the heap when possible. For more information about this option, see "-Xsoftmx" on page 168.

If the **-Xsoftmx** option is used, additional information is added to the MEMINFO section of a Javadump to indicate the target memory for the heap. See "Storage Management (MEMINFO)" on page 73.

Improved diagnostic information about Java threads

The THREADS section of a Javadump contains information about threads and stack traces. For Java threads, the thread ID and daemon status from the Java thread object is now recorded to help you diagnose problems. For more information, see "Threads and stack trace (THREADS)" on page 76.

Service refresh 5

Support is now available for 2 GB large pages on z/OS systems. New options are also provided for tuning and serviceability.

Setting default hardware prefetch behavior on AIX

A new command-line option, **-XXsetHWPrefetch:os-default**, is available for reverting to the default hardware prefetch behavior. Use this option to override a **-XXsetHWPrefetch:none** setting that you previously specified on the command line. For more information, see "-XXsetHWPrefetch:[none | os-default] (AIX only)" on page 154.

Change to default behavior for -Xcompressedrefs

The **-Xcompressedrefs** option is now enabled by default when the value of the **-Xmx** option is less than or equal to 25 GB, for all 64-bit operating systems other than z/OS. Use the **-Xnocompressedrefs** option to revert to the previous behavior. For z/OS operating systems, or values of **-Xmx** that are greater than 25 GB, compressed references are still disabled by default. For more information about these options, see JVM command-line options in the Java 6 Diagnostics Guide.

Support for dynamic machine configuration changes

A new command-line option, **-Xtune:elastic**, is available to turn on JVM function at run time that accommodates dynamic machine configuration changes. For more information, see "-Xtune" on page 151.

Support for 2 GB large pages on z/OS

On z/OS V1.13 with the RSM Enablement Offering, you can now request the JVM to allocate the Java object heap with 2 GB nonpageable large pages, by using the **-Xlp:objectheap** option. This option is supported only on the 64-bit SDK for z/OS, and requires certain prerequisites, which are described in the "Configuring large

page memory allocation" on page 37 topic. For more information about the RSM Enablement Offering, see http://www-03.ibm.com/systems/z/os/zos/downloads/#RSME.

Setting the JIT code cache page size

To be consistent with other platforms, the **-Xlp:codecache:pagesize=<size>** option is added for AIX and Linux PPC. As the code cache page size is derived from the operating system on these platforms, the page size can be altered only by changing operating system settings. The **-verbose:sizes** output shows the current page size. For more information about the **-Xlp:codecache:pagesize=<size>** option, see "-Xlp" on page 146.

New `-Xcheck:dump` option

This option runs AIX and Linux operating system checks during JVM startup. Messages are issued if the operating system has dump options or limits set that might truncate system dumps. This option is not supported on Windows or z/OS.

Service refresh 6

There is a change in behavior for the close() method of the FileInputStream. In addition, this service refresh provides improved diagnostic information to aid problem determination.

- "File descriptors are now closed immediately"
- "Operating system process information added to a Javadump file"

File descriptors are now closed immediately

There is a change to the default behaviour of the close() method of the FileInputStream, FileOutputStream, and RandomAccessFile classes. In previous releases, the default behavior was to close the file descriptor only when all the streams that were using it were also closed. The new default behavior is to close the file descriptor regardless of any other streams that might still be using it. You can revert to the previous default behavior by using a system property, however this property will be removed in future releases. For more information, see **-Dcom.ibm.streams.CloseFDWithStream** in the diagnostic guide for Version 6.

Operating system process information added to a Javadump file

The `ENVINFO` section contains a new line, `1CIPROCESSID`, which shows the ID of the operating system process that produced the core file.

See "TITLE, GPINFO, and ENVINFO sections" on page 70 for an example.

Service refresh 7

This service refresh provides options for securing Java API for XML (JAXP) processing. There is also a change to default fonts in z/OS V2.1, and several serviceability improvements.

- "Controlling JVM signal handling for CTRL_LOGOFF_EVENT" on page 12
- "Securing Java API for XML (JAXP) processing against malformed input" on page 12
- "Increasing the maximum size of the JIT code cache" on page 12
- "New path in font configuration properties file for z/OS" on page 12

Controlling JVM signal handling for CTRL_LOGOFF_EVENT

There is a new system property to control the way the JVM handles a CTRL_LOGOFF_EVENT signal when the JVM is running as an interactive Windows service. Setting this property to `true` prevents the JVM ending when the signal is received. For more information, see http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/topic/com.ibm.java.doc.diagnostics.60/diag/appendixes/cmdline/Dcomibmsignalhandlingignorelogoff.html.

Securing Java API for XML (JAXP) processing against malformed input

If your application takes untrusted XML, XSD or XSL files as input, you can enforce specific limits during JAXP processing to protect your application from malformed data. These limits can be set on the command line by using system properties, or you can specify values in your `jaxp.properties` file. You must also override the default XML parser configuration for the changes to take effect. For more information about configuring JAXP processing, see the Developing section of the User Guide for IBM SDK, Java Technology Edition, Version 6.

Increasing the maximum size of the JIT code cache

You can increase the maximum size of the JIT code cache by using a new system property. You cannot decrease the maximum size below the default value. For more information, see **`-Xcodecachetotal`** in the diagnostic guide for IBM SDK, Java Technology Edition, Version 6.

New path in font configuration properties file for z/OS

From z/OS V2.1, fonts are provided by the operating system. The paths to the font files in the `$JRE_LIB/fontconfig.properties.src` file have changed accordingly. If you have z/OS V2.1 or later, you do not have to install font packages or edit this properties file.

If you have z/OS V1.13 or earlier, you must now install font packages in the `/usr/lpp/fonts/worldtype` directory, or edit the properties file. For more information, see http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/topic/com.ibm.java.doc.user.zos.60/user/zos_fonts.html.

Thread CPU time information added to a Javadump file

For Java threads and attached native threads, the `THREADS` section contains, depending on your operating system, a new line: `3XMCPUTIME`. This line shows the number of seconds of CPU time that was consumed by the thread since that thread was started. For more information, see “Threads and stack trace (THREADS)” on page 76.

Support included for Windows releases

From this release, support is included for Windows 8.1 and Windows Server 2012 R2.

Service refresh 8

This service refresh provides new options.

- "JVM signal handling for SIGXFSZ"
- "Thread trace history in Java dump files"
- "Improvements to native memory and thread information in system dumps"
- "Securing Java API for XML (JAXP) processing against malformed input"
- "Improved diagnostic information for debugging wild branch problems"
- "CORBA debug enhancement" on page 14

JVM signal handling for SIGXFSZ

There is a new JVM option that allows the JVM to handle the operating system signal `SIGXFSZ` on the Linux platform. This signal is generated when a process attempts to write to a file that causes the maximum file size `ulimit` to be exceeded. If the signal is not handled by the JVM, the operating system ends the process with a core dump. This option is not enabled by default. For more information, see "-XX:[+|-]handleSIGXFSZ" on page 153.

Thread trace history in Java dump files

If your Java dump file was triggered by an exception throw, catch, uncaught, or systhrow event, or by the com.ibm.jvm.Dump API, the dump file contains recent trace history for the current thread. For more information, see "Trace history for the current thread" on page 82.

Improvements to native memory and thread information in system dumps

Additional information is now available in system dumps to help with problem determination. You can use dump viewer commands to find information about native memory and information about running threads. For more information, see "Commands available in **`jdmpview`**" on page 94. You can also obtain this information by using the DTFJ API. For more information, see the DTFJ API reference.

Securing Java API for XML (JAXP) processing against malformed input

You can control whether external entities are resolved in an XML document. This limit can be set on the command line by using a system property, or you can specify a value in your `jaxp.properties` file. You must also override the default XML parser configuration for the changes to take effect. For more information about configuring JAXP processing, see the Developing section of the User Guide for IBM SDK, Java Technology Edition, Version 6.

Improved diagnostic information for debugging wild branch problems

On z/OS and Linux on z Systems™, a new register called Break Event Address (BEA) is introduced into the `GPINFO` section of a javadump file, which stores the address of the last taken branch. The BEA register is useful for debugging wild branch problems, helping you to reconstruct the control flow paths that lead up to a crash. For more information about the `GPINFO` section, see "TITLE, GPINFO, and ENVINFO sections" on page 70.

CORBA debug enhancement

The extract() method of IBM ORB-generated Helper classes now throws an org.omg.CORBA.BAD_OPERATION exception if the type of the supplied Any object does not match the type that is expected by the Helper class. Previously, this method threw an org.omg.CORBA.MARSHAL exception or an OutOfMemoryError exception.

Service refresh 8 fix pack 1

This fix pack extends operating system support and includes minor changes and fixes to the program code.

- "Support for new operating systems"
- "Change to Zambian currency code symbol"
- "Comparator function"

Support for new operating systems

Support for Red Hat Enterprise Linux (RHEL) 7 is now available with this release.

Change to Zambian currency code symbol

In this release the currency code symbol for Zambia is corrected from "ZMK" to "ZMW".

Comparator function

A new system property is available to switch between the Java SE 6 and Java SE 5.0 implementation of the Comparator function. For more information, see "-Djava.util.Arrays.useLegacyMergeSort" on page 138.

Service refresh 8 fix pack 2

This fix pack includes new command-line options.

- "Specify a directory for all dump file types"
- "Changes to the access permissions for shared class caches (AIX, Linux, and z/OS operating systems only)"
- "Support for SUSE Linux Enterprise Server (SLES) 12" on page 16

Specify a directory for all dump file types

You can specify a directory to write all types of dump file to by using the new **-Xdump:directory** command-line option. This option enhances the existing ability to specify the dump file location for a particular dump agent type by using the **-Xdump:**<*agent*>**:file** option. You can use tokens in the **directory** option in the same way as in the **file** option. For more information about the **-Xdump** option, see "Using the -Xdump option" on page 63.

Changes to the access permissions for shared class caches (AIX, Linux, and z/OS operating systems only)

To enhance security, the virtual machine now runs access checks when a process attempts to access a shared class cache on the AIX, Linux, and z/OS operating systems. These checks are in addition to the existing checks that are run by the operating system on file access (for persistent caches) and System V objects (for

non-persistent caches). The access permissions for persistent shared class caches (not supported on z/OS) have also been modified, and are now the same as for non-persistent caches.

After the virtual machine runs the access checks, it grants or denies access as follows:

- Access is granted to the user that created the cache.
- Access is granted to any other user that is in the same group as the cache creator, but only if the **-Xshareclasses:groupAccess** option is specified on the command line.
- Access is denied in all other cases. For example, even if the cache has read permission for all, access is denied unless one of the previous points also applies.

Note: These access checks are not run for shared cache utility options such as **-Xshareclasses:printStats**, **-Xshareclasses:destroy**, or **-Xshareclasses:destroyAll**.

The following table summarizes the changes in access permissions for persistent caches on AIX and Linux operating systems:

Table 1. Changes to the access permissions for persistent shared class caches

Use of the -Xshareclasses:groupAccess command-line option at cache creation	Previous access permissions	New access permissions (now the same as non-persistent caches)
Not specified	`-rw-r--r--` R • Read/write for user, plus read-only for group and others	`-rw-------` • Read/write for user only
Specified	`-rw-rw-r--` • Read/write for user and group, plus read-only for others	`-rw-rw----` • Read/write for user and group only

You can revert to the previous behavior by specifying the **-Xshareclasses:cacheDir** option on the command line. When you use this option, the virtual machine does not run any access checks, so you must ensure that the specified directory has suitable access controls. Persistent caches are created with the same permissions as in the previous release.

These changes are likely to affect users in the following situations:

- A user in a group creates a cache by using the **-Xshareclasses:groupAccess** option, then another user in the same group attempts to access the cache without using the **-Xshareclasses:groupAccess** option. In this situation, access is now denied. The second user must specify the **-Xshareclasses:groupAccess** option.
- On AIX and Linux only: a user attempts to access a persistent cache that was created by another user in a different user group, by using the **-Xshareclasses:readonly** option. Read-only access for *group* and *other* categories has been removed, so access is now denied. To enable access in this situation,

create the cache by using the **-Xshareclasses:cacheDir** option, and set the permissions on the specified directory to allow read-only access to users who are outside the group of the cache creator.

For more information about shared classes command-line options, see "-Xshareclasses" on page 155.

Support for SUSE Linux Enterprise Server (SLES) 12

Support for this operating system is now available on certain platform architectures. For more information, see Chapter 4, "Hardware and software requirements," on page 31.

Service refresh 8 fix pack 3

This fix pack includes new command-line options.

- "Reserving memory space for compressed references"
- "Ability to turn off the ALT-key function"

Reserving memory space for compressed references

A new option is available for securing space in memory for any native classes, monitors, and threads that are used by compressed references. Setting this option can help prevent OutOfMemoryError exceptions that might occur if the lowest 4 GB of address space becomes full. For more information, see "-Xmcrs" on page 165.

Ability to turn off the ALT-key function

A new system property is available that can prevent the ALT-key from highlighting the first menu in the active window. For more information, see "-Dibm.disableAltProcessor" on page 138.

Note: If your application uses a Windows Look and Feel (com.sun.java.swing.plaf.windows.WindowsLookAndFeel), this option has no effect.

Service refresh 8 fix pack 4

This fix pack includes serviceability improvements.

- "Improvements to Java dump output"
- "Improved tracing for the Object Request Broker (ORB)"
- "Support for new operating systems" on page 17

Improvements to Java dump output

The THREADS section now contains more information about Java threads that are running, which helps with problem determination. For more information about these changes, see "Threads and stack trace (THREADS)" on page 76.

Improved tracing for the Object Request Broker (ORB)

Component level tracing is now available to improve the debugging of ORB problems. A new system property allows you to generate trace information for one or more ORB components, such as DISPATCH or MARSHAL. For more information about this system property, see

"-Dcom.ibm.CORBA.Debug.Component" on page 137.

Support for new operating systems

Support for Red Hat Enterprise Linux 7.1 is now available with this release. For a list of supported operating systems, see Chapter 4, "Hardware and software requirements," on page 31.

Service refresh 8 fix pack 7

This fix pack includes serviceability improvements.

- "Invalidating AOT methods in the shared classes cache"
- "Support for Windows 10"

Invalidating AOT methods in the shared classes cache

You can now invalidate failing AOT methods in the shared classes cache to prevent them being loaded without destroying and re-creating the cache. Three new **-Xshareclasses** suboptions are available to find, invalidate, or revalidate these methods. For more information, see "-Xshareclasses" on page 155.

Support for Windows 10

Microsoft Windows 10 is now supported.

Service refresh 8 fix pack 15

This fix pack includes extended operating system support and serviceability improvements.

- "Ability to check for the use of large pages"
- "AIX V7.2 support"
- "Support for z/OS V2.2"
- "Support for KVM for z Systems V1.1"
- "New system property -Djdk.xml.max.xmlNameLimit" on page 18

Ability to check for the use of large pages

You can now check whether large pages are obtained for the object heap when they are requested by the **-Xlp:objectheap:pagesize** option. The `warn` suboption generates a warning message if large pages are not obtained and allows the process to continue. Alternatively, you can use the `strict` suboption to generate an error message and end the process if large pages are not obtained. For more information, see "-Xlp" on page 146.

AIX V7.2 support

This release now supports AIX V7.2 on POWER7® and later processors.

Support for z/OS V2.2

This release now supports z/OS V2.2.

Support for KVM for z Systems V1.1

Support is now included for this virtualization software on z Systems.

New system property -Djdk.xml.max.xmlNameLimit

If your application takes untrusted XML, XSD or XSL files as input, you can enforce specific limits during JAXP processing to protect your application from malformed data. The **-Djdk.xml.max.xmlNameLimit** option can be used to limit the length of XML names in XML documents. For more information about this property, see Securing Java API for XML processing (JAXP) against malformed input.

Service refresh 8 fix pack 20

This fix pack includes a serviceability improvement for Windows, as well as Oracle and IBM fixes to the code base.

- "Improved diagnostic content in Windows Java VM dumps"
- "Change to default behavior for memory protection of a shared classes cache"

Improved diagnostic content in Windows Java VM dumps

Additional dump flags are set in the Java VM when system dumps are triggered on the Windows operating system. For more information, see: "System dumps" on page 65.

Change to default behavior for memory protection of a shared classes cache

On Linux and Windows platforms, new options are available to protect partially filled pages in the shared classes cache. These options prevent accidental memory overwrites, which can cause cache corruption. After the startup phase, a VM now protects partially filled pages by default. For more information about these options and the default setting, see the `mprotect` sub option in "-Xshareclasses" on page 155.

Service refresh 8 fix pack 25

This release contains Oracle and IBM fixes to the code base. In addition, the JVM creates a new shared classes cache, which means that existing shared caches can be removed.

Changes to the shared classes cache generation number

The format of classes that are stored in the shared classes cache is changed due to an Oracle security update. As a result, the shared cache generation number is changed, which causes the JVM to create a new shared classes cache, rather than re-creating or reusing an existing cache. To save space, all existing shared caches can be removed unless they are in use by an earlier release. For more information about deleting a shared classes cache, see "-Xshareclasses" on page 155.

Service refresh 8 fix pack 30

This release contains Oracle and IBM fixes to the code base.

Service refresh 8 fix pack 35

This release contains Oracle and IBM fixes to the code base.

Service refresh 8 fix pack 40

This release contains Oracle and IBM fixes to the code base.

Chapter 2. Understanding the IBM Software Developers Kit (SDK) for Java

You can read about the components of the SDK in the Diagnostics Guide for IBM SDK, Java Technology Edition, Version 6. Supplementary information for this release is included here.

A new Garbage Collection policy is available with this release of the SDK. Read the following section to learn about this policy and when to use it.

Balanced Garbage Collection policy

The Balanced Garbage Collection policy uses a region-based layout for the Java heap. These regions are individually managed to reduce the maximum pause time on large heaps, and also benefit from Non-Uniform Memory Architecture (NUMA) characteristics on modern server hardware.

The Balanced Garbage Collection policy is intended for environments where heap sizes are greater than 4 GB. The policy is available only on 64-bit platforms. You activate this policy by specifying **-Xgcpolicy:balanced** on the command line.

The Java heap is split into potentially thousands of equal sized areas called "regions". Each region can be collected independently, allowing the collector to focus only on the regions which return the largest amount of memory for the least processing effort.

Objects are allocated into a set of empty regions that are selected by the collector. This area is known as an "eden space". When the eden space is full, the collector stops the application to perform a Partial Garbage Collection (PGC). The collection might also include regions other than the eden space, if the collector determines that these regions are worth collecting. When the collection is complete, the application threads can proceed, allocating from a new eden space, until this area is full. This process continues for the life of the application.

From time to time, the collector starts a Global Mark Phase (GMP) to look for more opportunities to reclaim memory. Because PGC operations see only subsets of the heap during each collection, abandoned objects might remain in the heap. This issue is like the "floating garbage" problem seen by concurrent collectors. However, the GMP runs on the entire Java heap and can identify object cycles that are inactive for a long period. These objects are reclaimed.

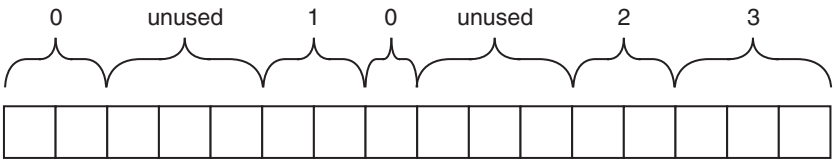
Region age

Age is tracked for each region in the Java heap, with 24 possible generations.

Like the Generational Concurrent Garbage Collector, the Balanced Garbage Collector tracks the age of objects in the Java heap. The Generational Concurrent Garbage Collector tracks object ages for each individual object, assigning two generations, "new" and "tenure". However, the Balanced Garbage Collector tracks object ages for each region, with 24 possible generations. An age 0 region, known as the "eden space", contains the newest objects allocated. The highest age region represents a maximum age where all long-lived objects eventually reside. A Partial

Garbage Collection (PGC) must collect age 0 regions, but can add any other regions to the collection set, regardless of age.

This diagram shows a region-based Java heap with ages and unused regions:



Note: There is no requirement that similarly aged regions are contiguous.

NUMA awareness

The Balanced Garbage Collection policy can increase application performance on large systems that have Non-Uniform Memory Architecture (NUMA) characteristics.

NUMA is used in multiprocessor systems on x86 and IBM POWER® architecture platforms. In a system that has NUMA characteristics, each processor has local memory available, but can access memory assigned to other processors. The memory access time is faster for local memory. A NUMA node is a collection of processors and memory that are mutually close. Memory access times within a node are faster than outside of a node.

The Balanced Garbage Collection policy can split the Java heap across NUMA nodes in a system. Application threads are segregated such that each thread has a node where the thread runs and favors the allocation of objects. This process increases the frequency of local memory access, increasing overall application performance.

A Partial Garbage Collection (PGC) attempts to move objects closer to the objects and threads that refer to them. In this way, the working set for a thread is physically close to where it is running.

The segregation of threads is expected to improve application performance. However, there might be some situations where thread segregation can limit the ability of an application to saturate all processors. This issue can result in slight fragmentation, slowing performance. You can test whether this optimization is negatively affecting your application by turning off NUMA awareness using the **-Xnuma:none** command-line option.

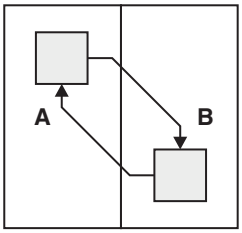
Partial Garbage Collection

A Partial Garbage Collection (PGC) reclaims memory by using either a Copy-Forward or Mark-Compact operation on the Java heap.

When the eden space is full, the application is stopped. A PGC runs before allocating another set of empty regions as the new eden space. The application can then proceed. A PGC is a “stop-the-world” operation, meaning that all application threads are suspended until it is complete. A PGC can be run on any set of regions in the heap, but always includes the eden space, used for allocation since the previous PGC. Other regions can be added to the set based on factors that include age, free memory, and fragmentation.

Because a PGC looks only at a subset of the heap, the operation might miss opportunities to reclaim dead objects in other regions. This problem is resolved by a Global Mark Phase (GMP).

In this example, regions A and B each contain an object that is reachable only through an object in the other region:



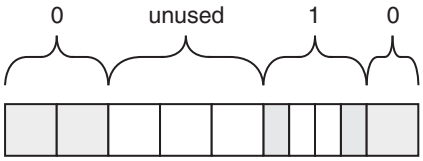
If only A or B is collected, one half of the cycle keeps the other alive. However, a GMP can see that these objects are unreachable.

The Balanced policy can use either a Copy-Forward (scavenge) collector or a Mark-Compact collector in the PGC operation. Typically, the policy favors Copy-Forward but can change either partially or fully to Mark-Compact if the heap is too full. You can check the verbose Garbage Collection logs to see which collection strategy is used.

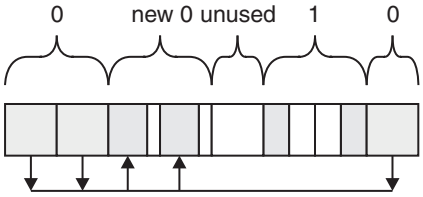
Copy-Forward operation

These examples show a PGC operation using Copy-Forward, where the shaded areas represent live objects, and the white areas are unused:

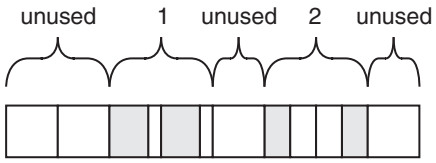
This diagram shows the Java heap before the Copy-Forward operation:



This diagram shows the Java heap during the Copy-Forward operation, where the arrows show the movement of objects:



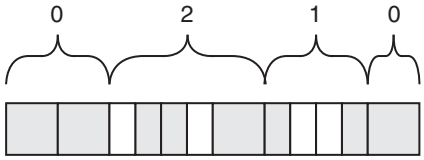
This diagram shows the Java heap after the Copy-Forward operation, where region ages have been incremented:



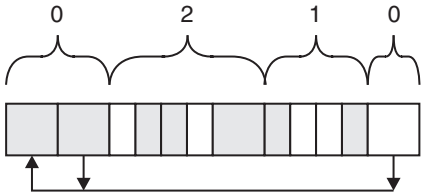
Mark-Compact operation

These examples show a PGC operation using Mark-Compact, where the shaded areas represent live objects, and the white areas are unused.

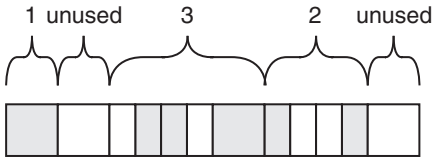
This picture shows the Java heap before the Mark-Compact operation:



This diagram shows the Java heap during the Mark-Compact operation, where the arrows show the movement of objects:



This diagram shows the Java heap after the Mark-Compact operation, where region ages have been incremented:



Global Mark Phase

A Global Mark Phase (GMP) takes place on the entire Java heap, finding, and marking abandoned objects for garbage collection.

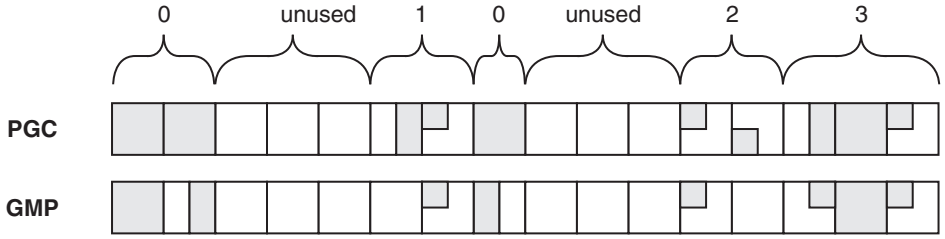
A GMP runs independently between Partial Garbage Collections (PGCs). Although the GMP runs incrementally, like the PGC, the GMP runs only a mark operation. However, this mark operation takes place on the entire Java heap, and does not make any decisions at the region level. By looking at the entire Java heap, the GMP can see more abandoned objects than the PGC might be aware of. The GMP does not start and finish in the same “stop-the-world” operation, which might lead

to some objects being kept alive as “floating garbage”. However, this waste is bounded by the set of objects that died after a given GMP started.

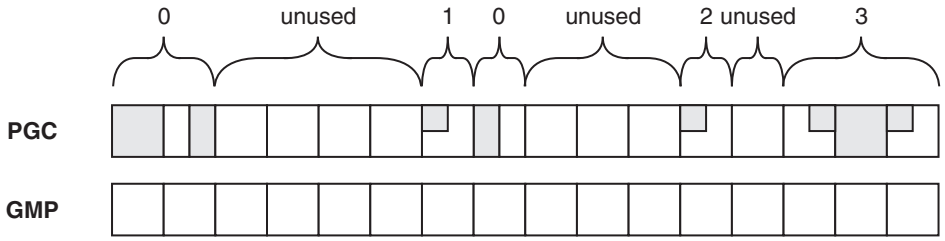
GMP also performs some work concurrently with the application threads. This concurrent mark operation is based purely on background threads, which allows idle processors to complete work, no matter how quickly the application is allocating memory. This concurrent mark operation is unlike the concurrent mark operations that are specified with **-Xgcpolicy:gencon** or **-Xgcpolicy:optavgpause**. For more information about the use of concurrent mark with these options, see ../../../../com.ibm.java.doc.diagnostics.60/diag/understanding/mm_gc_mark_concurrent.html.

When the GMP completes, the data that the PGC process is maintaining is replaced. The next PGC acts on the latest data in the Java heap.

This diagram shows that the GMP live object set is a subset of the PGC live object set when the GMP completes:



When the GMP replaces the data for use by the PGC operation, the next PGC uses this smaller live set for more aggressive collection. This process enables the GMP to clear all live objects in the GMP set, ready for the next global mark:



When to use the Balanced garbage collection policy

There are a number of situations when you should consider using the Balanced garbage collection policy. Generally, if you are currently using the Gencon policy, and the performance is good but the application still experiences large global collection (including compaction) pause times frequently enough to be disruptive, consider using the Balanced policy.

Note: Tools such as the IBM Monitoring and Diagnostic Tools - Garbage Collection and Memory Visualizer and IBM Monitoring and Diagnostic Tools - Health Center do not make recommendations that are specific to the Balanced policy.

Requirements

- This policy is available only on 64-bit platforms. The policy is not available if the application is deployed on 32-bit or 31-bit hardware or operating systems, or if the application requires loading 32-bit or 31-bit native libraries.
- The policy is optimized for larger heaps; if you have a heap size of less than 4 GB you are unlikely to see a benefit compared to using the Gencon policy.

Performance implications

The incremental garbage collection work that is performed for each collection, and the large-array-allocation support, cause a reduction in performance. Typically, there is a 10% decrease in throughput. This figure can vary, and the overall performance or throughput can also improve depending on the workload characteristics, for example if there are many global collections and compactions.

When to use the policy

Consider using the policy in the following situations:

The application occasionally experiences unacceptably long global garbage collection pause times

The policy attempts to reduce or eliminate the long pauses that can be experienced by global collections, particularly when a global compaction occurs. Balanced garbage collection incrementally reduces fragmentation in the heap by compacting part of the heap in every collection. By proactively tackling the fragmentation problem in incremental steps, which immediately return contiguous free memory back to the allocation pool, Balanced garbage collection eliminates the accumulation of work that is sometimes incurred by generational garbage collection.

Large array allocations are frequently a source of global collections, global compactions, or both

If large arrays, transient or otherwise, are allocated so often that garbage collections are forced even though sufficient total free memory remains, the Balanced policy can reduce garbage collection frequency and total pause time. The incremental nature of the heap compaction, and internal JVM technology for representing arrays, result in minimal disruption when allocating "large" arrays. "Large" arrays are arrays whose size is greater than approximately 0.1% of the heap.

Other areas that might benefit

The following situations might also benefit from use of this policy:

The application is multi-threaded and runs on hardware that demonstrates NUMA characteristics

Balanced garbage collection exploits NUMA hardware when multi-threaded applications are present. The JVM associates threads with NUMA nodes, and favors object allocation to memory that is associated with the same node as the thread. Balanced garbage collection keeps objects in memory that is associated with the same node, or migrates objects to memory that is associated with a different node, depending on usage patterns. This level of segregation and association can result in increased heap fragmentation, which might require a slightly larger heap.

The application is unable to use all the processor cores on the machine

Balanced garbage collection includes global tracing operations to break

cycles and refresh whole heap information. This behavior is known as the Global Mark Phase. During these operations, the JVM attempts to use under-utilized processor cores to perform some of this work while the application is running. This behavior reduces any stop-the-world time that the operation might require.

The application makes heavy use of dynamic class loading (often caused by heavy use of reflection)

The Gencon garbage collection policy can unload unused classes and class loaders, but only at global garbage collection cycles. Because global collection cycles might be infrequent, for example because few objects survive long enough to be copied to the tenure or old space, there might be a large accumulation of classes and class loaders in the native memory space. The Balanced garbage collection policy attempts to dynamically unload unused classes and class loaders on every partial collect. This approach reduces the time these classes and class loaders remain in memory.

When not to use the policy

The Java heap stays full for the entire run and cannot be made larger

The Balanced policy uses an internal representation of the object heap that allows selective incremental collection of different areas of the heap depending on where the best return on cost of garbage collection might be. This behavior, combined with the incremental nature of garbage collection, which might not fully collect a heap through a series of increments, can increase the amount of *floating garbage* that remains to be collected. Floating garbage refers to objects which might have become garbage, but which the garbage collector has not been able to immediately detect. As a result, if heap configurations already put pressure on the garbage collector, for example by resulting in little space remaining, the Balanced policy might perform poorly because it increases this pressure.

Real-time-pause guarantees are required

Although the Balanced policy typically results in much better worst-case pause time than the Gencon policy, it does not guarantee what these times are, nor does it guarantee a minimum amount of processor time that is dedicated to the application for any time window. If you require real-time guarantees, use a real-time product such as the IBM WebSphere Real Time product suite.

The application uses many large arrays

An array is "large" if it is larger than 0.1% of the heap. The Balanced policy uses an internal representation of large arrays in the JVM that is different from the standard representation. This difference avoids the high cost that the large arrays otherwise place on heap fragmentation and garbage collection. Because of this internal representation, there is an additional performance cost in using large arrays. If the application uses many large arrays, this performance cost might negate the benefits of using the Balanced policy.

Chapter 3. Migrating from earlier IBM SDK or runtime environments

This migration information applies to IBM SDK, Java Technology Edition, Version 6 (J9 VM 2.6).

If you are migrating from IBM SDK, Java Technology Edition, Version 6, read the following significant changes:

- Connections to virtual machines through the Java Attach API have a new default state. For more information, see Chapter 7, “Developing applications,” on page 43.
- Shared class caches are now persistent by default on the AIX operating system. For more information, see “-Xshareclasses” on page 155.
- The JIT compiler can use more than one thread to convert method bytecodes into native code, dynamically. If the default number of threads that are chosen by the JVM is not optimum for your environment, you can configure the number of threads by setting a system property. For more information, see “Using more than one JIT compilation thread” on page 48.
- The default garbage collection policy in force is now the Generational Concurrent garbage collector. For an overview, see ../../../com.ibm.java.doc.diagnostics.60/diag/understanding/mm_gc_generational.html.
- New optimizations for Java technology monitors are available, that are expected to improve CPU efficiency. If you experience performance problems that you suspect are connected to this release, see “Application performance issues” on page 59.
- Verbose garbage collection logging is redesigned. See “Verbose garbage collection logging” on page 111.
- The default value for **-Xjni:arrayCacheMax** is increased from 8096 bytes to 128 KB. Because more memory is used, this change might lead to an out of memory error.
- If you are migrating from a release of IBM SDK, Java Technology Edition, Version 6 before service refresh 16 fix pack 1, the currency code symbol for Zambia is now corrected to the value “ZMW”.
- For Linux on z System platforms, if you are migrating your hardware to IBM z13 you must install IBM SDK, Java Technology Edition, Version 6 Service Refresh 8 Fix Pack 3 or later to avoid a performance degradation.

New features and capabilities, which might present planning considerations, can be found here: “What's new” on page 1.

If you are migrating from the IBM SDK, Java 2 Technology Edition, Version 5.0, read the additional migration information available for IBM SDK, Java Technology Edition, Version 6 in the Information Center: http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/welcome/welcome_javasdk_version.html. A migration topic is included with each platform-specific user guide.

Chapter 4. Hardware and software requirements

There are changes to the supported hardware and operating system levels when IBM SDK, Java Technology Edition, Version 6 uses the IBM J9 2.6 virtual machine. Support is removed for Pentium 3 hardware, older Linux versions, and Windows 2000 server.

Any new updates to these support statements can be found in the Current news technote.

IBM SDK for AIX

The 32-bit and 64-bit SDKs run on hardware that supports the following platform architectures:

- IBM POWER 4
- IBM POWER 5
- IBM POWER 6
- IBM POWER 7
- IBM POWER8®
- JS20 blades

The SDKs also run on older System p systems that have a Common Hardware Reference Platform (CHRP) architecture. To test whether the SDK is supported on a specific System p system, at the system prompt type:

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

The output for a supported platform reads:

```
Model Architecture: chrp
```

The following table shows the operating systems supported for each platform architecture. The table also indicates whether support for an operating system release was included at the "general availability" (GA) date for the SDK, or at a specific service refresh (SR) level:

Table 2. AIX environments tested

Operating system	32-bit SDK	64-bit SDK
AIX 6.1.0.4	GA	GA
AIX 7.1.0.0	GA	GA
AIX 7.2.0.0	SR8 FP15	SR8 FP15

Note: AIX 7.2 is supported only on IBM POWER 7 and later processors.

IBM SDK for Linux

There are a number of distributions provided for the Linux operating system that support the following platform architectures:

- Intel Architecture, 32-bit (IA-32)
  - Pentium 4

  - Pentium Xeon
  - Pentium M
  - Pentium D and equivalents
- AMD64/EM64T
- IBM POWER 32
- IBM POWER 64
- z Systems 31-bit
- z Systems 64-bit

The following z Systems are supported:

- IBM z13™ [1]
- IBM zEnterprise BC12
- IBM zEnterprise EC12
- IBM zEnterprise 196
- IBM zEnterprise 114
- z10™
- IBM System z9® [2]
- IBM System z990 [2]
- IBM System z900 [2]
- IBM System z800 [2]

Notes:

1. If you are migrating from EC12, z196, z10, or z9 systems to a z13 system, you must update to service refresh 8 fix pack 3 to avoid a performance degradation.
2. These products are withdrawn from marketing.

The following table shows the operating systems supported for each platform architecture. The table also indicates whether support for an operating system release was included at the "general availability" (GA) date for the SDK, or at a specific service refresh (SR) or fix pack (FP) level:

Table 3. Linux environments tested

Hardware	**IA-32 32-bit**	**AMD64/EM64T 64-bit**		**POWER 64-bit**		**z Systems 31-bit**	**z Systems 64-bit**	
SDK address space	*32-bit*	*32-bit*	*64-bit*	*32-bit*	*64-bit*	*31-bit*	*31-bit*	*64-bit*
SLES 10 Service pack 3	GA	GA	GA	GA	GA	GA	GA	GA
SLES 11	GA	GA	GA	GA	GA	GA	GA	GA
SLES 12	SR8 FP2	SR8 FP2	SR8 FP2	-	-	SR8 FP2	SR8 FP2	SR8 FP2
RHEL 5 Update 6	GA	GA	GA	GA	GA	GA	GA	GA
RHEL 6	GA	GA	GA	GA	GA	GA	GA	GA
RHEL 7	SR8 FP1	SR8 FP1	SR8 FP1	SR8 FP1	SR8 FP1	SR8 FP1	SR8 FP1	SR8 FP1
RHEL 7.1	SR8 FP4	SR8 FP4	SR8 FP4	SR8 FP4	SR8 FP4	SR8 FP4	SR8 FP4	SR8 FP4

Table 3. Linux environments tested (continued)

Hardware	IA-32 32-bit	AMD64/EM64T 64-bit		POWER 64-bit		z Systems 31-bit	z Systems 64-bit	
Ubuntu 8.04	GA	GA	GA	-	-	-	-	-
Ubuntu 10.04	GA	GA	GA	-	-	-	-	-
Ubuntu 12.04	SR8	SR8	SR8	-	-	-	-	-
Ubuntu 14-04	SR8	SR8	SR8	-	-	-	-	-

Note: On SLES 11 SP1, an intermittent problem is seen that causes the Java process to end with the error `res_query.c:251: __libc_res_nquery: Assertion ``hp != hp2\' failed.`. If you have a SUSE customer services contract, you can obtain a fix for this problem by quoting the SUSE bug number 747932.

Note: On an IA-32 platform architecture with SLES 10 service pack 2, the Java process might hang. This problem is not seen with SLES 10 service pack 3.

IBM SDK for Windows

The 32-bit SDK for Windows runs on hardware that supports the Intel 32-bit architecture. The following hardware is supported:

- Pentium 4
- Pentium Xeon
- Pentium M
- Pentium D and equivalents

The 64-bit SDK for Windows runs on hardware that supports the AMD64 or EM64T architecture.

The following table shows the operating systems supported for each platform architecture. The table also indicates whether support for an operating system release was included at the "general availability" (GA) date for the SDK, or at a specific service refresh (SR) level:

Table 4. Windows environments tested

Operating system	32-bit SDK	64-bit SDK
Windows XP service pack 3	GA	GA
Windows Vista service pack 2	GA	GA
Windows 7	GA	GA
Windows 8	SR4	SR4
/Windows 10	SR8 FP7	SR8 FP7
Windows Server 2003 service pack 1	GA	GA
Windows Server 2003 R2	GA	GA
Windows Server 2008 service pack 2	GA	GA

Table 4. Windows environments tested (continued)

Operating system	32-bit SDK	64-bit SDK
Windows Server 2008 R2 service pack 1	GA	GA
Windows Server 2012	SR4	SR4
Windows Server 2012 R2	SR7	SR7

IBM SDK for z/OS

The z/OS 31-bit and 64-bit SDKs run on the following z Systems:

- IBM z13
- IBM zEnterprise BC12
- IBM zEnterprise EC12
- IBM zEnterprise 196
- IBM zEnterprise z114
- IBM System z10
- IBM System z9 (see note)
- IBM System z990 (see note)
- IBM System z900 (see note)
- IBM System z800 (see note)

Note: These products are withdrawn from marketing

The following table shows the operating systems supported for each platform architecture. The table also indicates whether support for an operating system release was included at the "general availability" (GA) date for the SDK, or at a specific service refresh (SR) level:

Table 5. z/OS environments tested

Operating system	31-bit SDK	64-bit SDK	Comments
z/OS 1.10	GA	GA	Operating system support ended 2011.
z/OS 1.11	GA	GA	Operating system support ended 2012.
z/OS 1.12	GA	GA	Operating system support ended 2014.
z/OS 1.13	SR1	SR1	Operating system support ends 30 September 2016.
z/OS 2.1	SR8	SR8	
z/OS 2.2	SR8 FP15	SR8 FP15	

Virtualization software

For information about the virtualization software tested, see Support for virtualization software.

Chapter 5. Installation

Read this section to learn about any important installation changes that apply to IBM SDK, Java Technology Edition, Version 6 with a IBM J9 2.6 virtual machine.

This section provides information that is supplemental to the information about installing and configuring the IBM SDK, Java Technology Edition, Version 6, located at http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/welcome/welcome_javasdk_version.html.

Setting the path

The **PATH** environment variable you use for IBM SDK, Java Technology Edition, Version 6, Release 0, Modification 1 must be set correctly.

About this task

On z/OS, the installation directory for IBM SDK, Java Technology Edition, Version 6, Release 0, Modification 1 is different from earlier releases. You must alter your **PATH** environment variable so that z/OS can find programs and utilities, such as **javac**, **java**, and **javadoc** tool from any current directory.

To display the current value of your **PATH**, type the following command at a command prompt:

```
echo
$PATH
```

Set the **PATH** environment variable according to your platform:

- For 31-bit z/OS:

  ```
  export PATH=/usr/lpp/java/J6.0.1/bin:/usr/lpp/java/J6.0.1/bin:$PATH
  ```

- For 64-bit z/OS:

  ```
  export PATH=/usr/lpp/java/J6.0.1_64/bin:/usr/lpp/java/J6.0.1_64/bin:$PATH
  ```

Chapter 6. Running Java technology applications

Applications can be started using the launcher or through JNI. Settings are passed to an application using command-line arguments, environment variables, and properties files.

The information provded here applies only to this release. General information for IBM SDK, Java Technology Edition, Version 6 can be found in the User Guides and Diagnostic Guide that are available in the IBM Information Center: http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/welcome/welcome_javasdk_version.html.

Configuring large page memory allocation

You can enable large page support, on systems that support it, by starting the Java process with the **-Xlp** option.

Large page usage is primarily intended to provide performance improvements to applications that allocate a great deal of memory and frequently access that memory. The large page performance improvements are a result of the reduced number of misses in the Translation Lookaside Buffer (TLB). The TLB maps a larger virtual storage area range and thus causes this improvement.

AIX

AIX requires special configuration to enable large pages. For more information about configuring AIX support for large pages, see

AIX 6.1
http://publib.boulder.ibm.com/infocenter/aix/v6r1/topic/com.ibm.aix.prftungd/doc/prftungd/large_page_ovw.htm

AIX 7.1
http://publib.boulder.ibm.com/infocenter/aix/v7r1/topic/com.ibm.aix.prftungd/doc/prftungd/large_page_ovw.htm

The SDK supports the use of large pages only to back the Java object heap shared memory segments. The JVM uses shmget() with the SHM_LGPG and SHM_PIN flags to allocate large pages. The **-Xlp** option replaces the environment variable **IBM_JAVA_LARGE_PAGE_SIZE**, which is now ignored if set.

For the JVM to use large pages, your system must have an adequate number of contiguous large pages available. If large pages cannot be allocated, even when enough pages are available, possibly the large pages are not contiguous.

To obtain the large page sizes available and the current setting, use the **-verbose:sizes** option. Note the current settings are the requested sizes and not the sizes obtained. For object heap size information, check the **-verbose:gc** output.

For more information about the **-Xlp** option, see “-Xlp” on page 146.

Linux

Large page support must be available in the kernel, and enabled, so that the JVM can use large pages.

To configure large page memory allocation, first ensure that the running kernel supports large pages. Check that the file `/proc/meminfo` contains the following lines:

```
HugePages_Total:     <number of pages>
HugePages_Free:      <number of pages>
Hugepagesize:        <page size, in kB>
```

The number of pages available and their sizes vary between distributions.

If large page support is not available in your kernel, these lines are not present in the `/proc/meminfo` file. In this case, you must install a new kernel containing support for large pages.

If large page support is available, but not enabled, `HugePages_Total` has the value 0. In this case, your administrator must enable large page support. Check your operating system manual for more instructions.

For the JVM to use large pages, your system must have an adequate number of contiguous large pages available. If large pages cannot be allocated, even when enough pages are available, possibly the large pages are not contiguous. Configuring the number of large pages at boot up creates them contiguously.

Large page allocations only succeed if the user is a member of the group with its gid stored in `/proc/sys/vm/hugetlb_shm_group`, or if you run the Java process with root access. See the huge page support in the Linux kernel documentation for more information.

If a non-root user needs access to large pages, their locked memory limit must be increased. The locked memory limit must be at least as large as the maximum size of the Java heap. The maximum size of the Java heap can be specified using the **-Xmx** command-line option, or determined by adding **-verbose:sizes** and inspecting the output for the value `-Xmx`. If **pam** is not installed, change the locked memory limit using the **ulimit** command. If **pam** is installed, change the locked memory limit by adding the following lines to `/etc/security/limits.conf`:

```
@<large group name> soft memlock 2097152
@<large group name> hard memlock 2097152
```

Where *<large group name>* is the name of the group with its gid stored in `/proc/sys/vm/hugetlb_shm_group`.

To obtain the large page sizes available and the current setting, use the **-verbose:sizes** option. Note the current settings are the requested sizes and not the sizes obtained. For object heap size information, check the **-verbose:gc** output.

For more information about the **-Xlp** option, see For more information about the **-Xlp** option, see "-Xlp" on page 146.

Windows

To use large pages, the user that runs the Java process must have the authority to "lock pages in memory". To enable this authority, as Administrator go to **Control**

Panel > **Administrative Tools** > **Local Security Policy** and then find **Local Policies** > **User Rights Assignment** > **Lock pages in memory**. Alternatively, run **secpol.msc**. Add the user who runs the Java process, and reboot your machine. For more information, see these websites:

- http://msdn.microsoft.com/en-us/library/aa366720(VS.85).aspx
- http://msdn.microsoft.com/en-us/library/aa366568(VS.85).aspx

For the JVM to use large pages, your system must have an adequate number of contiguous large pages available. If large pages cannot be allocated, even when enough pages are available, possibly the large pages are not contiguous.

Large page allocations only succeed if the local administrative policy for the JVM user has the **Lock pages in memory** setting enabled.

On Microsoft Windows Vista and later, and Windows 2008 and later, use of large pages is affected by the User Account Control (UAC) feature. When UAC is enabled, a regular user (a member of the Users group) can use the **-Xlp** option as normal. However, an administrative user (a member of the Administrators group) must run the application as an administrator to gain the privileges required to lock pages in memory. To run as administrator, right-click the application and click **Run as administrator**. If the user does not have the necessary privileges, the following error message is produced: `System configuration does not support option '-Xlp'`.

To obtain the large page sizes available and the current setting, use the **-verbose:sizes** option. Note the current settings are the requested sizes and not the sizes obtained. For object heap size information, check the **-verbose:gc** output.

For more information about the **-Xlp** option, see "-Xlp" on page 146.

z/OS

Sub-options are available to request the JVM to allocate the Java object heap or the JIT code cache using large pages. These options are shown in the table, together with the large page sizes supported.

Table 6. Large page size support. Large page sizes supported for **-Xlp** options

Large page size	-Xlp:codecache	-Xlp:objectheap	-Xlp
2G nonpageable	Not supported	Supported (64-bit JVM only)	Supported (64-bit JVM only)
1M nonpageable	Not supported	Supported (64-bit JVM only)	Supported (64-bit JVM only)
1M pageable	Supported (31-bit and 64-bit JVM)	Supported (31-bit and 64-bit JVM)	Not supported

For more information about the **-Xlp** option, see "-Xlp" on page 146.

The following restrictions apply to large page sizes on z/OS:

2G nonpageable

- This page size applies to object heap large pages. The JIT code cache cannot be allocated in 2GB nonpageable large pages.
- This page size is supported only on the 64-bit SDK for z/OS, not the 31-bit SDK.

- This page size requires z/OS V1.13 with PTFs and the z/OS V1.13 Remote Storage Manager Enablement Offering web deliverable, and an IBM zEnterprise EC12 processor or later.
- A system programmer must configure z/OS for 2G nonpageable large pages.
- Users who require large pages must be authorized to the IARRSM.LRGPAGES resource in the RACF® (or an equivalent security product) FACILITY class with read authority.

1M nonpageable

- This page size applies to object heap large pages. The JIT code cache cannot be allocated in 1M nonpageable large pages.
- This page size is supported only on the 64-bit SDK for z/OS, not the 31-bit SDK.
- This page size requires z/OS V1.10 with APAR OA25485, and a System z10® processor or later.
- A system programmer must configure z/OS for 1M nonpageable large pages.
- Users who require large pages must be authorized to the IARRSM.LRGPAGES resource in the RACF (or an equivalent security product) FACILITY class with read authority.

1M pageable

- This page size is supported on the 31-bit and 64-bit SDK for z/OS.
- Both the object heap and the JIT code cache can be allocated in 1M pageable large pages.
- The use of 1M pageable pages for the object heap provides similar runtime performance benefits to the use of 1M nonpageable pages. In addition, using 1M pageable pages provides options for managing memory that can improve system availability and responsiveness.
- The following minimum prerequisites apply: IBM zEnterprise EC12 with the Flash Express feature (#0402), z/OS V1.13 with PTFs, APAR OA41307, and the z/OS V1.13 Remote Storage Manager Enablement Offering web deliverable.

When the JVM is allocating large pages, if a particular large page size cannot be allocated, the following sizes are attempted, in order, where applicable:

- 2G nonpageable
- 1M nonpageable
- 1M pageable
- 4K pageable

For example, if 1M nonpageable large pages are requested but cannot be allocated, pageable 1M large pages are attempted, and then pageable 4K pages.

The option **PAGESCM=ALL | NONE** in the IEASYSxx parmlib member controls 1M pageable large pages for the entire LPAR. ALL is the default. Therefore, when you run your application on a z/OS system that supports Flash, and that has Flash cards installed, the Flash card is available for paging by default. As a result, RSM also allows the use of 1M pageable large pages.

The option **LFAREA** in the IEASYxx parmlib member controls both 2G nonpageable and 1M nonpageable large pages for the entire LPAR. You can use the z/OS system command `DISPLAY VS,LFAREA` to show **LFAREA** usage information for the

entire LPAR. For more information, see the documentation for your version of z/OS. For example: http://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.ieae100/lfarea.htm?lang=en.

To obtain the large page sizes available and the current setting, use the **-verbose:sizes** option. Note the current settings are the requested sizes and not the sizes obtained. For object heap size information, check the **-verbose:gc** output.

Specifying a heap size that is a multiple of the page size uses another page of memory. For large sizes like 2G, you should set the heap size smaller than the next page size boundary. For example, when using the 2G pagesize, specify a maximum heap size of **-Xmx2047m** instead of **-Xmx2048m**, or **-Xmx4095m** instead of **-Xmx4096m**, and so on. When using nonpageable large pages, the real memory size that you specify is allocated when the JVM starts. For example, using options **-Xmx1023m -Xms512m -Xlp:objectheap:pagesize=1M,nonpageable** allocates 1G of real memory for the 1M nonpageable pages when the JVM starts.

All platforms

When specifying **-Xmx** or **-Xms**, the physical storage allocated is based on the page size. For example, if using 2G large pages with Java options **-Xmx1024M** and **-Xms 512K**, the Java heap is allocated on a 2G large page. The real memory for the 2G large page is allocated immediately. Even though the Java heap is consuming a 2G large page, in this example, the maximum Java heap is 1024M with an initial Java heap of 512K as specified. If the 2G pagesize is not pageable, the 2G large page is never paged out as long as the JVM is running. For more information about the **-Xmx** option, see "-Xmx" on page 167.

Chapter 7. Developing applications

This section contains important considerations for developing Java applications.

JRIO

The JRIO component available in earlier versions of the IBM SDKs for z/OS has been supplanted by the increasing functionality and enhancements of the JZOS component.

In IBM SDK, Java Technology Edition, Version 6, Release 0, Modification 1, the JRIO component is deprecated. Existing JRIO functions continue to be supported, but compiling Java source code that references JRIO classes causes warnings that identify occurrences of deprecated classes.

As an alternative, use the record I/O facilities provided in the JZOS component. For more information about JZOS, see: Java Batch Launcher and Toolkit for z/OS. In applications that use JRIO classes, search the source code for references to the package:

```
import com.ibm.recordio;
```

The presence of this package identifies source code containing references to JRIO classes.

For service refresh 1, a tracking macro is included with the product that can be used to determine if and where applications are using JRIO functions. For more information, see: IBM Java Record I/O (JRIO).

Java Attach API

The status of the Java Attach API support depends on the release of the product, and your operating system:

Table 7. Default status of the Attach API

Release	All platforms, except z/OS	z/OS 31-bit and 64-bit
Service refresh 3 and earlier	Enabled by default	Enabled by default
Service refresh 4 and later	Enabled by default	Disabled by default

If support is enabled, you must secure access to the Attach API function to ensure that only authorized users or processes can connect to another virtual machine. If you do not intend to use the capability, disable it using the following Java system property:

```
-Dcom.ibm.tools.attach.enable=no
```

If support is disabled, you can enable it by specifying the following system property, after ensuring secure access:

```
-Dcom.ibm.tools.attach.enable=yes
```

For more information about the Attach API, see the Version 6 information center.

Chapter 8. Debugging

You can debug applications by using the Java Debugger (JDB) application and various utilities that are provided with the SDK.

A platform-specific JDB is provided with the SDK that can be used to debug your Java applications. There are also a number of tools provided with the SDK that you can use to analyze JVM components, such as the shared classes cache.

The selective debugging feature, enabled using the command-line option **-XselectiveDebug**, is no longer supported with the IBM J9 2.6 virtual machine.

ORB debug property

New values were added for the `com.ibm.CORBA.Debug` property in version 6, service refresh 11. These new values are not supported in IBM SDK Java Technology Edition Version 6 with the IBM J9 2.6 virtual machine. For more information, see the diagnostic guide for IBM SDK Java Technology Edition Version 6.

Chapter 9. Performance

This performance information applies only to IBM SDK, Java Technology Edition, Version 6 (J9 VM 2.6).

General information for IBM SDK, Java Technology Edition, Version 6 can be found in the User Guides and Diagnostic Guide that are available in the IBM Information Center: http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/welcome/welcome_javasdk_version.html.

This release introduces new optimizations for Java technology monitors that are expected to improve CPU efficiency. New locking optimizations are also implemented that are expected to reduce memory usage and improve performance. If you experience performance problems that you suspect are connected to this release, see "Application performance issues" on page 59.

Garbage collection policy options

The **-Xgcpolicy** options control the behavior of the Garbage Collector. There are a number of changes to garbage collection options.

The changes to the garbage collection options are:

-Xgcpolicy:gencon
This option is now the default for garbage collection. The policy requests the combined use of concurrent and generational garbage collection to help minimize the time that is spent in any garbage collection pause.

-Xgcpolicy:optthruput
This option is no longer the default for garbage collection. The policy delivers high throughput to applications, but at the cost of occasional pauses.

-Xgcpolicy:subpool
This option is deprecated and is now an alias for **optthruput**. Therefore, if you use this option, the effect is the same as **optthruput**.

-Xgcpolicy:balanced
The balanced garbage collection policy is new. This policy uses a region-based layout for the Java heap. These regions are individually managed to reduce the maximum pause time on large heaps and increase the efficiency of garbage collection. The policy also uses a different object allocation strategy that improves application throughput on large systems that have Non-Uniform Memory Architecture (NUMA) characteristics. (x86 and POWER platforms only) For more information about this policy, see "Balanced Garbage Collection policy" on page 21.

For a detailed description of the garbage collection options available with IBM SDK, Java Technology Edition, Version 6, see Garbage collection options.

Changes to command-line options used by the Garbage Collector are detailed in "Garbage collection command-line options" on page 164.

Tuning implications for the Balanced garbage collection policy

Moving from a different garbage collection policy to the Balanced policy can affect any tuning that you made under the old policy.

New space settings might need adjusting when moving from the Gencon policy

If you change from using the Gencon policy to the Balanced policy, you might need to reduce the amount of new space (specified using the **-Xmn** parameter). The Gencon policy stores all "new" and "young" objects in a subset of the space reserved as new space. Although the Gencon policy changes how it defines "young" objects to suit its needs, the limit on the amount of memory it will try to collect in one new space collection is always limited by the new space size. The Balanced policy, however, stores only "new" objects in the space reserved as new space. This space is the strict minimum amount of memory that the Balanced policy must collect in every partial collect, but the policy will add other "young", "nearly empty", or "fragmented" regions of the heap to its collection set in a given partial collect, if it determines that it must in order to maintain a steady state.

Applications that saturate all processors with running threads might affect Balanced pause times

Balanced garbage collection attempts to reduce stop-the-world garbage collection pause times by performing some operations concurrently with running Java threads. If no processor time is available for the garbage collection work, then the time and frequency of stop-the-world pauses can increase. At the same time, the concurrent garbage collection worker threads might cause running Java threads to be paused, slightly impacting throughput. If the hardware processor resources are not fully used by the application, the concurrency aspects of the Balanced policy run optimally.

Balanced total native memory footprint is larger than other configurations

For the same Java heap size, the Balanced policy uses more native memory than other garbage collection policies, including the Gencon policy, due to additional metadata structures that it requires. Typically, this extra storage is approximately 6-7% of the Java heap size.

The Balanced policy might require a larger Java heap than the same workload in other garbage collection policies

The Balanced policy can ignore areas of the heap which it determines are not going to yield much free memory, however there is typically some memory available there. This small amount of wasted memory can accumulate over the entire length of the heap. Additionally, between global mark phases, the Balanced policy cannot break some kinds of cyclic reference chains. This behavior means that the policy might determine that some objects are still in use even though they are not. These two factors result in additional Java heap consumption.

Using more than one JIT compilation thread

The JIT compiler can use more than one thread to convert method bytecodes into native code, dynamically.

The JIT compiler can use more than one compilation thread to perform JIT compilation tasks. Using multiple threads can potentially help Java applications to start, or ramp-up, faster. In practice, multiple JIT compilation threads show performance improvements only where there are unused processing cores in the system.

The default number of compilation threads is identified by the JVM, and is dependent on the system configuration. If the resulting number of threads is not optimum, you can override the JVM decision by using the "-XcompilationThreads" on page 143 option.

Note: If your system does not have unused processing cores, increasing the number of compilation threads is unlikely to produce a performance improvement.

Chapter 10. Security

Learn about any important security changes that apply to IBM SDK, Java Technology Edition, Version 6 (J9 VM 2.6). These changes are common to IBM SDK, Java Technology Edition, Version 6, which contains J9 VM 2.4.

For security information that relates to the following SDK packages, click the links:

- IBM 31-bit SDK for z/OS Java Technology Edition, Version 6.0.1
- IBM 64-bit SDK for z/OS Java Technology Edition, Version 6.0.1

For security information about other platforms, including example code, see Security reference for IBM SDK Java Technology Edition, Version 6. For a short summary of the changes in each update, read the following sections:

Note: Although the changes are common to IBM SDK, Java Technology Edition, Version 6, which contains J9 VM 2.4, the service refresh levels differ. If you use the full security reference guide, Security reference for IBM SDK Java Technology Edition, Version 6, read the sections to determine the equivalent service refresh levels.

Initial release

In the first release of this product, the security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 9. The following change applies for security support:

PKCS11 security provider Cryptographic Support

The following card is supported in a limited fashion on the AIX platform, in both 32-bit and 64-bit modes: The IBM 4765 PCIe Cryptographic Coprocessor is supported for use only by Tivoli® Key Lifecycle Manager release 2.0.1, and follow-on releases.

For Tivoli Key Lifecycle Manager, only the following PKCS#11 crypto-operations are supported:

- Translate an AES 128-bit or 256-bit software key to an AES hardware (PKCS#11) key.
- Generate an AES 128-bit or 256-bit key.
- Encrypt and decrypt data by using an AES key, and an AES/ECB/NoPadding cipher.
- Store and retrieve an AES key to, or from, a PKCS11IMPLKS (PKCS#11) key store.

Service refresh 1

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 10. Key changes for security include NIST SP800-131a compliance. Support is provided for Transport Layer Security (TLS) 1.1 and 1.2 protocols, and elliptic curve and AES-GCM cipher suites.

Service refresh 2

There are no changes to the security component.

Service refresh 3

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 11.

The IBM Common Access Card (IBMCAC) provider enables applications to use standard APIs to access the United States Department of Defense Common Access Card (CAC). This provider is available only on the Windows platform.

For more information, see IBM Common Access Card provider.

The IBM PKCS#11 provider now supports the following cryptographic adapters:

- SafeNet Luna SA 4.0
- SafeNet Luna SA 5.0
- Thales nShield Edge
- Thales nShield Connect 1500

For points of interest about these adapters, see Card observations.

Service refresh 4

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 12.

The IBMJCEFIPS provider has been FIPS certified. If you use IBMJSSE2, this security provider can now be run in FIPS mode. For more information, see Running IBMJSSE2 in FIPS mode.

IKeyman is a GUI application that provides key, certification request and self-signed certification generation operations. The **ikeycmd** command is enhanced to show all certificates in the certificate chain. For more information, see iKeyman.

Service refresh 5

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 13.

A new set of policy files should be used for the JVM. Although the old policy files continue to work with all current releases, after installing service refresh 5, you should plan to update to the new policy files before 2014. This activity is necessary ahead of the expiry of the certificates that sign these policy files. For more information, see IBM SDK policy files.

Service refresh 6

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 14.

In previous releases, when you used the KeyGenerator class to generate DESede keys by using the IBMPKCS11Impl provider, the IBMPKCS11Impl provider supported only a DESede key size of 192. The IBMPKCS11Impl provider now also accepts a DESede key size of 168, to be consistent with the IBM JCE security provider, which accepts a DESede key size of 168. 168 and 192 actually represent the same DESede key size; 192 includes the DESede parity bits, but 168 does not.

Service refresh 7

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 15.

The following enhancement is made in JSSE: If you have not explicitly configured an SSL socket factory, you can use a system property to override the SSL protocol that is specified by the default SSL socket factory. For more information, see http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/com.ibm.java.security.component.60.doc/security-component/jsse2Docs/overrideSSLprotocol.html.

Service refresh 8

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 16

You can now specify a location for the unlimited jurisdiction policy files, instead of having to move the files to a specific directory within the SDK. By placing the files in a location that is outside the SDK, you ensure that the files are not overwritten when you upgrade the SDK. Use the **-Dcom.ibm.security.jurisdictionPolicyDir=<policy_file_location>** system property to specify the new location. For more information, see IBM SDK policy files.

Service refresh 8 fix pack 1

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 16, fix pack 1. There are no changes to the security component in this update.

Service refresh 8 fix pack 2

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 16, fix pack 2. The following change is made in this release:

SSL V3.0 protocol is disabled by default
To address the Padding Oracle On Downgraded Legacy Encryption (POODLE) security vulnerability, the SSL V3.0 protocol is disabled by default and TLS is enabled. As a result, there is a significant change in default behavior that will cause failures in any applications that rely on SSL V3.0. For more information, see IBM SDK, Java Technology Edition fixes to mitigate against the POODLE security vulnerability (CVE-2014-3566).

Change to PKCS11Impl supported algorithms
The Java algorithm Cipher.RSA/SSL/PKCS1Padding now uses the PKCS11 mechanism `CKM_RSA_PKCS` instead of the `CKM_RSA_X_509` mechanism. The Cipher.RSA/ECB/NoPadding algorithm and the Cipher.RSA/ /NoPadding algorithm now use the CKM_RSA_X_509 mechanism.

For the full list of supported algorithms, see PKCS11 supported algorithms.

iKeyman user guide
There is an updated version of the user guide available with this release. For more information, see Overview.

New system property for Transport Layer Security (TLS) renegotiation
To address Oracle security fix 8037066, a further system property, **`jdk.tls.allowUnsafeServerCertChange=`**`[false | true]`, is available. Use this property to allow unsafe server certificate change in renegotiation. For more information, see Transport Layer Services (TLS) renegotiation.

PKCS12 KeyStore changes
The IBM PKCS12 KeyStore implementation now has enhanced security for the storage of PrivateKey objects. PrivateKey objects are now stored in a ShroudedKeyBag, which is similar to the Oracle KeyStore implementation. ShroudedKeyBag objects are encrypted in addition to the file level encryption that protects the other contents of the KeyStore. KeyStores created with the previous version of the IBM PKCS12 KeyStore implementation can still be read by the newer implementation. However, storing new items in the KeyStore causes that KeyStore to be converted to the newer format with enhanced security.

Note: This change does not apply to KeyStore types PKCS12S and PKCS12S2.

Service refresh 8 fix pack 3

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 16, fix pack 3.

New Oracle security property affecting SSL V3.0

To address the POODLE security vulnerability, Oracle have introduced the security property **jdk.tls.disabledAlgorithms**, which is set to SSLv3 by default in the java.security file. This property takes precedence over the IBM system property that was implemented to address this vulnerability in service refresh 8 fix pack 2. For more information, see IBM SDK, Java Technology Edition fixes to mitigate against the POODLE security vulnerability (CVE-2014-3566).

Service refresh 8 fix pack 4

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 16, fix pack 4.

Factoring Attack on RSA-EXPORT keys (FREAK) security vulnerability

To address the security vulnerability CVE-2015-0138, **RSA_EXPORT** ciphers are no longer enabled by default.

RSA-PSS signature scheme

RSA-PSS is a new signature scheme that is based on the RSA cryptography system and provides enhanced security. PSS refers to the original Probabilistic Signature Scheme that was designed by Bellare and Rogaway.

To use this signature scheme, specify the algorithm name **"RSAPSS"** in the getInstance(algorithm, provider) method of the Signature class.

The PSSParameterSpec class specifies a parameter specification for the RSA-PSS signature scheme. You can set the following parameters:

- The algorithm name of the hash function (default SHA-1).
- The name of the mask generation function (default "MGF1").
- The parameters for the mask generation function (default MGF1ParameterSpec.SHA1).
- The length of salt (default *20*).
- The value of the trailer field (default *1*).

Note: The default values are the only supported values for the name of the mask generation function and trailer field.

By using one of the constructors for this class, you can construct a PSSParameterSpec class and then specify it as an argument to the set(PSSParameterSpec) method of the Signature class. This action sets the parameters for the RSA-PSS signature scheme. For more information about how to set parameters, see the PSSParameterSpec class documentation.

For more information about RSA-PSS, see RFC 3447.

RC4 cipher suites are disabled by default

To address security vulnerability CVE-2015-2808, RC4 cipher suites are disabled by default and Cipher-Block Chaining (CBC) protection is enabled. For more information, see Bar Mitzvah security vulnerability CVE-2015-2808.

Matching SSLv3 to SSL behavior

To address the POODLE security vulnerability, the SSL V3.0 protocol is disabled by default. If your application hardcodes the protocol label SSLv3, you can use the **com.ibm.jsse2.convertSSLv3** property to automatically match the behavior for protocol label SSL without modifying your source code. For more information about this property, see Matching SSLv3 to SSL behavior.

Service refresh 8 fix pack 5

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 16, fix pack 5.

Weak Diffie-Hellman (DH) keys are disabled by default
To address security vulnerability CVE-2015-4000, DH keys that are less than 768-bits are disabled by default. For more information, see Logjam security vulnerability CVE-2015-4000.

Service refresh 8 fix pack 7

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 16, fix pack 7.

Important changes to maintain FIPS 140-2 compliance
IBMJCEFIPS, Version 1.71 is FIPS certified and includes a fix for the reseeding of HASHDRBG. Two actions are necessary to maintain compliance beyond 2015:

1. All applications must install the updated versions of IBMJCEFIPS and IBM JSSE Provider JAR files (`ibmjcefips.jar` and `ibmjsseprovider2.jar`) that are provided in this fix pack.
2. Applications that call IBMSecureRandom must make a small code change to call HASHDRBG instead.

For more information about these changes, see IBM JCE FIPS 140-2 Cryptographic Module Security Policy.

Changes to the security property `jdk.tls.disabledAlgorithms`
This property **`jdk.tls.disabledAlgorithms`** now supports the disabling of cipher suites by naming the cryptographic algorithm to be disabled. The default value for this property is **`jdk.tls.disabledAlgorithms=SSLv3, RC4, DH keySize < 768`**.

Change to IBMJSSE2 behavior when a minimum DH key size is not set in the `java.security` file
If the `java.security` file is not updated with `DH keySize < 768` for the **`jdk.tls.disabledAlgorithms`** property, IBMJSSE2 applies a minimum default key size of 768 for DH keys. For more information, see Logjam security vulnerability CVE-2015-4000.

Service refresh 8 fix pack 15

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 16, fix pack 15.

The following changes are made as a result of Oracle security updates:

New security property `com.ibm.security.krb5.autodeducerealm=true|false`
This property is set to `false` by default. A security permission check is performed on a principal with deduced realm. The check ensures that only the authorized principal can initiate or accept secure connections. If the value of this property is `true`, there is no security check performed.

Ability to customize the Ephemeral Diffie-Hellman key size
Diffie-Hellman (DH) keys of sizes less than 1024 bits are deprecated because of their insufficient strength. You can now customize the

ephemeral DH key size with the system property jdk.tls.ephemeralDHKeySize. For more information, see Customizing the Ephemeral Diffie-Hellman key size.

New security property `jdk.tls.server.defaultDHEParameters`
JSSE uses a default set of hardcoded Diffie-Hellman (DH) primes for each DH group. To improve the security of DH key pair generation, you can now provide custom values for DH primes by using the security property **`jdk.tls.server.defaultDHEParameters`**, or by configuring the `java.security` file. For more information, see the `java.security` file.

Service refresh 8 fix pack 20

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 16, fix pack 20.

New support for RFC5915 encoded EC private keys
Support for EC private keys that are encoded according to the format specified in the RFC5915 document is added to the IBMJCE provider. The ibm.security.internal.spec.RFC5915ECPrivateKeyEncodedKeySpec class is introduced to represent these private keys.

Certificates signed with MD5 are no longer allowed by default
In response to the SLOTH security vulnerability, the use of MD5 in SSL communication is disabled in the SDK by default. Certificates signed with MD5 are no longer allowed. However, if you are unable to use an alternative in the short term, you can reverse this change by making changes to the `java.security` file that is located in the `<JAVA_HOME>\lib\security` directory.

- Remove the value `MD5` from the property **`jdk.certpath.disabledAlgorithms`**.
- Remove the value `MD5withRSA` from the property **`jdk.tls.disabledAlgorithms`**.

Note: By removing these values and allowing the use of certificates signed with MD5 in SSL communication you are exposed to the SLOTH security vulnerability.

Service refresh 8 fix pack 25

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 16, fix pack 25.

The AES-GCM cipher algorithm internal initialization vector generation feature of the IBMJCE provider is improved to comply with the NIST SP 800-38D specification. The specification requires that the number of encryption operations, for a cipher instance by using the same encryption key, must be limited to a maximum allowable number of iterations. When that number of iterations is exhausted, an exception is thrown to notify the caller that a fresh encryption key must be used to reinitialize the cipher instance before encryption can continue.

The following exception is thrown: `IllegalStateException ("The maximum number of IV invocations for the current key have been exhausted.")`

The maximum number of iterations is 18,446,744,073,709,551,615.

Service refresh 8 fix pack 30

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 16, fix pack 30.

New version of the JCE FIPS provider

This release includes version 1.8 of the IBM JCE FIPS provider. There are behavior differences between this version and the previous version, 1.71. These differences might require you to modify your application code if you are using the IBMJCEFIPS provider or the JSSE provider in FIPS mode. For more information, see IBM JCE FIPS 140-2 Cryptographic Module Security Policy.

JGSS: Specifying Kerberos encryption types

You can now specify Kerberos encryption types by using the `com.ibm.security.krb5.enctypes` Java system property. For more information, see Some JGSS Used Java Properties.

Service refresh 8 fix pack 35

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 16, fix pack 35. There are no changes to the security component in this update.

Service refresh 8 fix pack 40

The security component is equivalent to IBM SDK, Java Technology Edition, Version 6, service refresh 16, fix pack 40.

Matching the behavior of SSLContext.getInstance("TLS") to Oracle

To match the behavior of SSLContext.getInstance("TLS") with the Oracle implementation, set the property **`com.ibm.jsse2.overrideDefaultTLS`** to true. The following protocols are enabled: TLSV1.0, V1.1, and V1.2. For more information, see Matching the behavior of SSLContext.getInstance("TLS") to Oracle.

Changes to default protocols in the IBMJSSE2 provider

By default, TLS V1.1 and V1.2 are now enabled on the client and server. For more information, see Protocols.

Changes to IBMJSSE2 cipher support

3DES is now considered to be a weak cipher and should not be used unless a stronger cipher is not available in the client requested cipher suites. The `DESede` algorithm is added to the list of algorithms that are disabled by default. For more information, see Disabling cryptographic algorithms.

Chapter 11. Troubleshooting and support

Troubleshooting and support.

Problem determination

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

JVM messages

IBM JVM messages can help you with problem determination. All JVM messages are written to the standard error (stderr) stream, and selected messages are written to the system log.

Messages are issued by the JVM in response to certain conditions, including warning and error situations. These messages can indicate the source of the problem, allowing you to take corrective action. Additional information for each JVM message, including suggested actions, is provided in the IBM J9 VM Messages guide, http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/welcome/welcome_javasdk_version.html.

By default, all error messages and some information messages are written to the system log. The specific information messages are `JVMDUMP006I`, `JVMDUMP032I`, and `JVMDUMP033I`, which provide valuable additional information about dumps produced by the JVM. However, you can use the **`-Xlog`** command-line option to configure the types of messages that are recorded. For more information about the **`-Xlog`** option, see "-Xlog" on page 145.

You can also control JVM message logging using JVMTI extensions for IBM. There are two new APIs available to query and modify the JVM settings. For more information, see "IBM JVMTI extensions" on page 126.

Application performance issues

Performance problems might be associated with new optimizations that have been introduced.

Java monitor optimizations

New optimizations are expected to improve CPU efficiency. However, there might be some situations where lower CPU utilization is achieved, but overall application performance decreases. You can test whether the new optimizations are negatively affecting your application by reverting to the behavior of earlier versions.

- If performance is affected as soon as you start using this release, use the following command-line option to revert to the old behavior.

  ```
  -Xthr:secondarySpinForObjectMonitors
  ```

  Use the following command-line option to reestablish the new behavior.

  ```
  -Xthr:noSecondarySpinForObjectMonitors
  ```

  Note: This optimization is not implemented on the AIX platform.

- If performance is affected after the application has run for some time, or after a period of heavy load, use the following command-line option to revert to the old behavior.

  ```
  -Xthr:noAdaptSpin
  ```

  Use the following command-line option to reestablish the new behavior.

  ```
  -Xthr:AdaptSpin
  ```

Linux operating systems only. If your application uses the Thread.yield() method extensively, it might be negatively affected by the optimizations of the default locking behavior on Linux systems that are using the Completely Fair Scheduler (CFS) in the default mode (`sched_compat_yield=0`). You can test whether these optimizations are negatively affecting your application by running the following test:

1. Use the following command-line option to revert to behavior that is closer to earlier versions and monitor application performance:

   ```
   -Xthr:noCfsYield
   ```

2. If performance does not improve, remove the previous command-line options or use the following command-line option to reestablish the new behavior:

   ```
   -Xthr:cfsYield
   ```

Lock optimizations

New locking optimizations are expected to reduce memory usage and improve performance. However, there might be some situations where a smaller heap size is achieved for an application, but overall application performance decreases. For example, if your application synchronizes on objects that are not typically synchronized on, such as Java.lang.String, run the following test:

Use the following command-line option to revert to behavior that is closer to earlier versions and monitor application performance:

```
-Xlockword:mode=all
```

If performance does not improve, remove the previous command-line option or use the following command-line option to reestablish the new behavior:

```
-Xlockword:mode=default
```

Receiving OutOfMemoryError exceptions

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

If the Java heap is exhausted, an error message is received indicating an OutOfMemoryError condition with the Java heap.

If the native heap is exhausted, an error message is received indicating that a native allocation failed.

In either case, the problem might not be a memory leak. The steady state of memory use that is required might be higher than the memory 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. However, this action does delay the onset of the OutOfMemoryError exception or error conditions, which might 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.

OutOfMemoryError exceptions on z/OS

The JVM throws a java.lang.OutOfMemoryError exception when the heap is full and the JVM cannot find space for object creation. Heap usage is a result of the application design, its use and creation of object populations, and the interaction between the heap and the garbage collector.

The operation of the JVM's Garbage Collector is such that objects are continuously allocated on the heap by mutator (application) threads until an object allocation fails. At this point, a garbage collection cycle begins. At the end of the cycle, the allocation is tried again. If successful, the mutator threads resume where they stopped. If the allocation request cannot be fulfilled, an out-of-memory exception occurs. See ../../../../com.ibm.java.doc.diagnostics.60/diag/understanding/memory_management.html for more detailed information.

An out-of-memory exception occurs when the live object population requires more space than is available in the Java managed heap. This situation can occur because of an object leak or because the Java heap is not large enough for the application that is running. If the heap is too small, you can use the **-Xmx** option to increase the heap size and remove the problem, as follows:

```
java -Xmx320m MyApplication
```

If the failure occurs under **javac**, remember that the compiler is a Java program itself. To pass parameters to the JVM that is created for compilation, use the **-J** option to pass the parameters that you normally pass directly. For example, the following option passes a 128 MB maximum heap to **javac**:

```
javac -J-Xmx128m MyApplication.java
```

In the case of a genuine object leak, the increased heap size does not solve the problem and also increases the time taken for a failure to occur.

Out-of-memory exceptions also occur when a JVM call to malloc() fails. This should normally have an associated error code.

If an out-of-memory exception occurs and no error message is produced, the Java heap is probably exhausted. To solve the problem:

- Increase the maximum Java heap size to allow for the possibility that the heap is not big enough for the application that is running.
- Enable the z/OS Heapdump.
- Turn on **-verbose:gc** output.

The **-verbose:gc** (**-verbose:gc**) switch causes the JVM to print out messages when a garbage collection cycle begins and ends. These messages indicate how much live

data remains on the heap at the end of a collection cycle. In the case of a Java object leak, the amount of free space on the heap after a garbage collection cycle decreases over time. See “Verbose garbage collection logging” on page 111.

A Java object leak is caused when an application retains references to objects that are no longer in use. In a C application you must free memory when it is no longer required. In a Java application you must remove references to objects that are no longer required, usually by setting references to null. When references are not removed, the object and anything the object references stays in the Java heap and cannot be removed. This problem typically occurs when data collections are not managed correctly; that is, the mechanism to remove objects from the collection is either not used or is used incorrectly.

The JVM produces a heap dump and a system dump when an OutOfMemoryError exception is thrown. Use a tool to analyze the dumps to find out why the Java heap is full. The recommended tool for analyzing the heap dump or system dump is the IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer, see the information center for IBM Monitoring and Diagnostic Tools for Java.

If an OutOfMemoryError exception is thrown due to private storage area exhaustion under the 31-bit JVM, verify if the environment variable *_BPX_SHAREAS* is set to NO. If *_BPX_SHAREAS* is set to YES multiple processes are allowed to share the same virtual storage (address space). The result is a much quicker depletion of private storage area. For more information on *_BPX_SHAREAS*, see the z/OS documentation in IBM Knowledge Center. For example: Setting _BPX_SHAREAS and _BPX_SPAWN_SCRIPT.

Tracing the Object Request Broker (ORB)

Properties to use to enable ORB traces.

You can turn on ORB tracing by using the **-Dcom.ibm.CORBA.Debug** system property. The following options can be used with this property:

false Tracing is not enabled, which is the default value.

true Tracing is enabled.

From service refresh 8 fix pack 4, a new property is available for debugging the ORB at the subcomponent level:

- **com.ibm.CORBA.Debug.Component**: This property generates trace output only for specific Object Request Broker (ORB) subcomponents. The following subcomponents can be specified:
  - DISPATCH
  - MARSHAL
  - TRANSPORT
  - CLASSLOADER
  - ALL

  When you want to trace more than one of these subcomponents, each subcomponent must be separated by a comma.

```
java -Dcom.ibm.CORBA.Debug=true -Dcom.ibm.CORBA.Debug.Output=trace.log
 -Dcom.ibm.CORBA.Debug.Component=DISPATCH -Dcom.ibm.CORBA.CommTrace=true <classname>
```

Attention: Do not enable tracing for normal operation, because it might cause a performance degradation. First Failure Data Capture (FFDC) still works when tracing is turned off, which means that serious errors are reported. If a debug file is produced, examine it for issues. For example, the server might have stopped without performing an ORB.shutdown().

For more information about ORB debug properties, see Debug properties.

Using diagnostic tools

Diagnostic tools are available to help you solve your problems.

The sections in this part are:

Using dump agents

Dump agents are set up during JVM initialization. There are additional dump agent events available with the IBM J9 2.6 virtual machine.

Using the -Xdump option

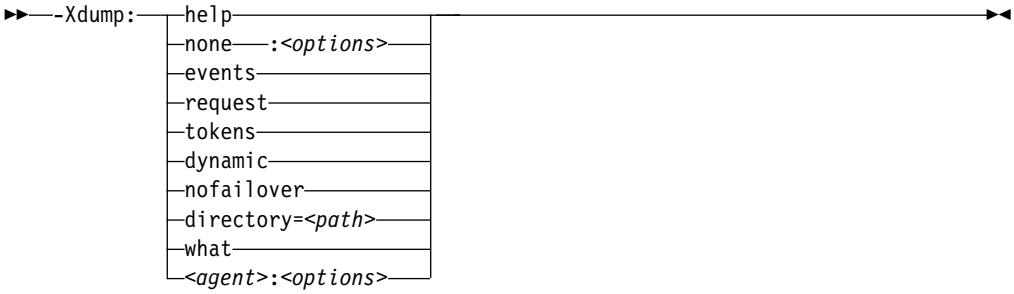
The **-Xdump** option controls the way you use dump agents and dumps.

You can use the **-Xdump** option 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.

The syntax of the **-Xdump** option is as follows:

-Xdump command-line option syntax



You can have multiple **-Xdump** options on the command line. You can also have multiple dump types triggered by multiple events. For example, the following command line turns off all Heapdumps, and creates a dump agent that produces a Heapdump and a Javadump when either a `vmstart` or `vmstop` event occurs:

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

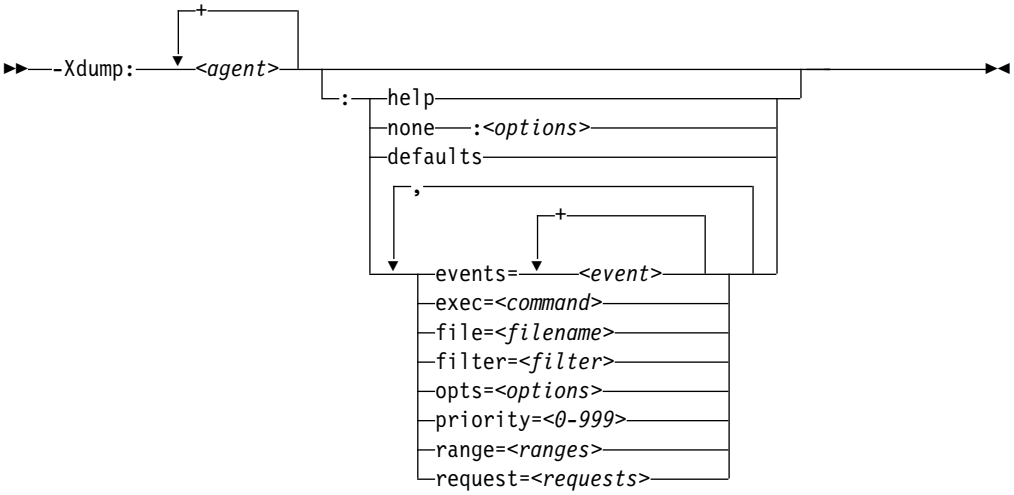
You can use the **-Xdump:directory=** option to specify a directory for all dump file types to be written to. This directory path is prefixed to the path of all non-absolute dump file names, including the file names for the default dump agents.

You can use the **-Xdump:what** option to list the registered dump agents. The registered dump agents listed might be different to the agents you specified. The difference is 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 for further control of the dump produced.

The options that you can use with dump agents provide granular control. The following syntax applies:

-Xdump command-line agent option syntax



Users of UNIX style shells must be aware that unwanted shell expansion might occur because of the characters used in the dump agent options. To avoid unpredictable results, enclose this command-line option in quotation marks. For example:

```
java "-Xdump:java:events=throw,filter=*Memory*" <Class>
```

For more information, see the manual for your shell.

Help options

These options provide 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	Provides detailed dump agent help
-Xdump:<*agent*>:defaults	Provides default settings for this agent

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 diagnostic data collection.

For a list of dump agents, see ../../../../com.ibm.java.doc.diagnostics.60/diag/tools/dumpagents_agents.html.

Supplementary information that applies to this release is included in the following sections:

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.

Windows:

```
java -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 'C:\sdk\sdk\jre\bin\my.dmp'
JVMDUMP010I System Dump written to C:\sdk\sdk\jre\bin\my.dmp
JVMDUMP013I Processed Dump Event "vmstop", detail "#00000000".
```

Other platforms:

```
java -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".
```

On z/OS, system dumps are written to data sets in the MVS™ file system. The following syntax is used:

```
java -Xdump:system:dsn=%uid.MVS.DATASET.NAME
```

Windows system dumps are created with the following options set: `MiniDumpWithFullMemory`, `MiniDumpWithHandleData`, `MiniDumpWithUnloadedModules`, `MiniDumpWithFullMemoryInfo` and `MiniDumpWithThreadInfo`. These options are equivalent to the options that are set when a dump is created by using the Windows task manager. The option definitions, and the full list of Windows dump flags, are documented in the Windows Dev Center.

See "Using the dump viewer" on page 89 for more information about analyzing a system dump.

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 by specifying **-Xdump:tokens**. If you do not specify a tool to use, a platform specific debugger is started.

On Windows, the following system output is seen:

```
java -Xdump:tool:events=vmstop,exec="cmd /c echo process %pid has finished" -version
....
JVMDUMP039I Processing dump event "vmstop", detail "#0000000000000000" at 2012/03/01
17:52:47 - please wait.
JVMDUMP007I JVM Requesting Tool dump using 'cmd /c echo process 2140 has finished'
JVMDUMP011I Tool dump created process 4996
process 2140 has finished
JVMDUMP013I Processed dump event "vmstop", detail "#0000000000000000".
```

On other platforms, the following system output is seen:

```
java -Xdump:tool:events=vmstop,exec="echo process %pid has finished" -version
...
JVMDUMP006I Processing dump event "vmstop", detail "#00000000" - please wait.
JVMDUMP007I JVM Requesting Tool dump using 'echo process 6620 has finished'
JVMDUMP011I Tool dump created process 6641
process 6620 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`.

By default, the thread that launches the external process waits for that process to end before continuing. The `opts` option can be used to modify this behavior.

Dump events

Dump agents are triggered by events occurring during JVM operation.

The following table shows the new events that are available:

Event	Triggered when...	Filter operation
corruptcache	The JVM finds that the shared class cache is corrupt.	Not applicable.
excessivegc	An excessive amount of time is being spent in the garbage collector	Not applicable.

For a list of other events available with Java 6, see ../../../../com.ibm.java.doc.diagnostics.60/diag/tools/dumpagents_events.html.

Default dump agents

The JVM adds a set of dump agents by default during initialization. You can override this set of dump agents using **-Xdump** on the command line.

By default, dump files are written to the virtual machine's current working directory. You can override this value by specifying the **-Xdump:directory** option at startup to specify a different dump directory.

There are additional dump events to those in the IBM J9 2.4 virtual machine. The registered dump agents and default dump events are shown in the output from the **-Xdump:what** command. This output varies according to platform.

On z/OS, the **-Xdump:system** output is changed for the event `systhrow`.

z/OS platform

```
Registered dump agents
----------------------
-Xdump:system:
    events=gpf+user+abort+traceassert+corruptcache,
    label=%uid.JVM.TDUMP.%job.D%y%m%d.T%H%M%S,
   range=1..0,
  priority=999,
  request=serial
----------------------
-Xdump:system:
    events=systhrow,
    filter=java/lang/OutOfMemoryError,
    label=%uid.JVM.TDUMP.%job.D%y%m%d.T%H%M%S,
    range=1..1,
    priority=999,
    request=exclusive+compact+prepwalk
----------------------
-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=500,
  request=exclusive+compact+prepwalk,
  opts=PHD
----------------------
-Xdump:java:
  events=gpf+user+abort+traceassert+corruptcache,
  label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
  range=1..0,
  priority=400,
  request=exclusive+preempt
----------------------
-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=400,
  request=exclusive+preempt
----------------------
-Xdump:snap:
  events=gpf+abort+traceassert+corruptcache,
  label=/home/user/Snap.%Y%m%d.%H%M%S.%pid.%seq.trc,
  range=1..0,
  priority=300,
  request=serial
----------------------
-Xdump:snap:
```

```
  events=systhrow,
  filter=java/lang/OutOfMemoryError,
  label=/home/user/Snap.%Y%m%d.%H%M%S.%pid.%seq.trc,
  range=1..4,
  priority=300,
  request=serial
----------------------
```

Other platforms

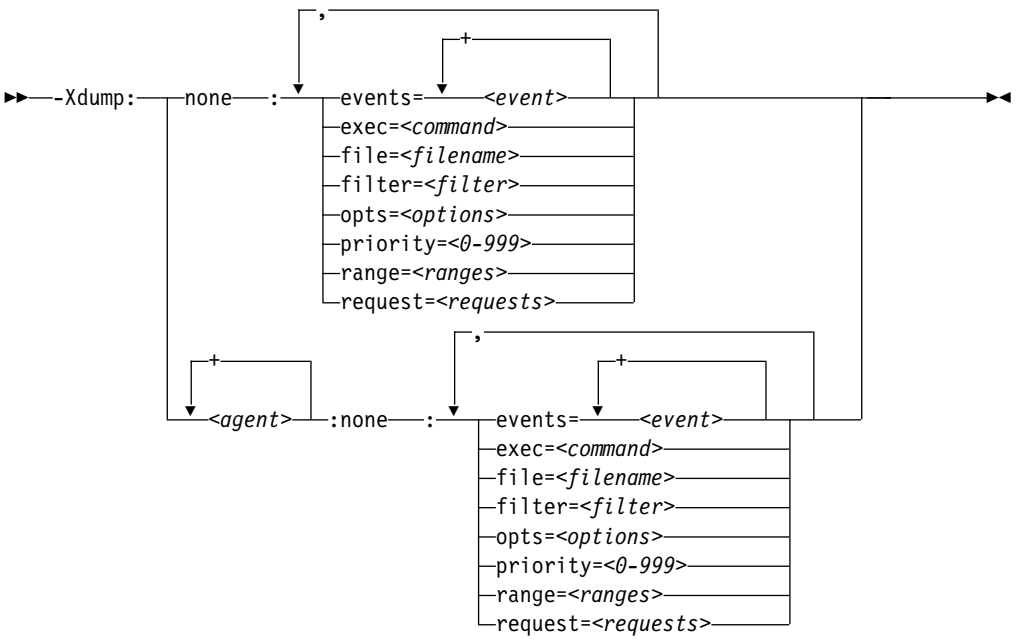
```
Registered dump agents
----------------------
-Xdump:system:
    events=gpf+abort+traceassert+corruptcache,
    label=/home/user/core.%Y%m%d.%H%M%S.%pid.%seq.dmp,
    range=1..0,
    priority=999,
    request=serial
----------------------
-Xdump:heap:
    events=systhrow,
    filter=java/lang/OutOfMemoryError,
    label=/home/user/heapdump.%Y%m%d.%H%M%S.%pid.%seq.phd,
    range=1..4,
    priority=500,
    request=exclusive+compact+prepwalk,
    opts=PHD
----------------------
-Xdump:java:
    events=gpf+user+abort+traceassert+corruptcache,
    label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
    range=1..0,
    priority=400,
    request=exclusive+preempt
----------------------
-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=400,
    request=exclusive+preempt
----------------------
-Xdump:snap:
    events=gpf+abort+traceassert+corruptcache,
    label=/home/user/Snap.%Y%m%d.%H%M%S.%pid.%seq.trc,
    range=1..0,
    priority=300,
    request=serial
----------------------
-Xdump:snap:
    events=systhrow,
    filter=java/lang/OutOfMemoryError,
    label=/home/user/Snap.%Y%m%d.%H%M%S.%pid.%seq.trc,
    range=1..4,
    priority=300,
    request=serial
----------------------
```

Removing dump agents

You can specify the `none` option with **`-Xdump`** to remove dump agents of a particular type or with particular settings.

The following syntax diagram shows you how you can use the `none` option:

-Xdump command-line syntax: the none option



You can remove all default dump agents and any preceding dump options by using:

```
-Xdump:none
```

Use this option if you want to specify a new dump configuration to ensure that previous settings are removed.

You can selectively remove dump agents, by event type, with the **-Xdump** option.

Here are some examples:

To turn off all Heapdumps (including default agents) but leave Javadump enabled, use the following option:

```
-Xdump:java+heap:events=vmstop -Xdump:heap:none
```

To turn off all dump agents for corruptcache events:

```
-Xdump:none:events=corruptcache
```

To turn off just system dumps for corruptcache events:

```
-Xdump:system:none:events=corruptcache
```

To turn off all dumps when `java/lang/OutOfMemory` error is thrown:

```
-Xdump:none:events=systhrow,filter=java/lang/OutOfMemoryError
```

To turn off just system dumps when `java/lang/OutOfMemory` error is thrown:

```
-Xdump:system:none:events=systhrow,filter=java/lang/OutOfMemoryError
```

If you remove all dump agents using **-Xdump:none** with no further **-Xdump** options, the JVM still provides these basic diagnostic outputs:

- 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. See http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/com.ibm.java.doc.diagnostics.60/diag/appendixes/cmdline/cmdline_specifying.html for more information on using a command-line options file.

Using Javadump

Javadump produces files that contain diagnostic information related to the JVM and a Java application captured at a point during run time. Javadump produces information about native memory use.

You can control the production of Javadumps by enabling dump agents, see http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/com.ibm.java.doc.diagnostics.60/diag/tools/dump_agents.html. You can also use environment variables, see http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/com.ibm.java.doc.diagnostics.60/diag/tools/javadump_env.html.

Default agents are in place that 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.

The content and range of information in a Javadump might change between JVM versions or service refreshes.

Note: Javadump is also known as Javacore. The default file name for a Javadump is `javacore.<date>.<time>.<pid>.<sequence number>.txt`. Javacore is NOT the same as a core file, which is generated by a system dump.

TITLE, GPINFO, and ENVINFO sections

The first three sections of a javadump provide useful information about the cause of a dump. The TITLE and ENVINFO sections are different in javadumps that are generated from an IBM J9 2.6 virtual machine.

TITLE

This section shows basic information about the event that caused the generation of the javadump, including the time and name. An additional line is included in the `TITLE` section that indicates the character set used in the javadump.

```
0SECTION       TITLE subcomponent dump routine
NULL           ===============================
1TICHARSET     850
1TISIGINFO     Dump Event "vmstart" (00000001) received
1TIDATETIME    Date: 2011/01/19 at 13:14:46
1TIFILENAME    Javacore filename: C:\test\javacore.20110119.131446.3808.0001.txt
1TIREQFLAGS    Request Flags: 0x81 (exclusive+preempt)
1TIPREPSTATE   Prep State: 0x6 (vm_access+exclusive_vm_access)
```

GPINFO

On z/OS and Linux on z Systems, the Break Event Address (BEA) register stores the address of the last branch taken in the GPINFO section. Here is a sample on z/OS:

```
0SECTION       GPINFO subcomponent dump routine
NULL           ================================
2XHOSLEVEL     OS Level         : z/OS 01.12.00
...
1XHREGISTERS   Registers:
2XHREGISTER      gpr0: FFFFFFFFFFFFFFFF
2XHREGISTER      gpr1: 00000048109FA088
2XHREGISTER      gpr2: FFFFFFFFFFFFFFFF
2XHREGISTER      gpr3: 0000000000000001
2XHREGISTER      gpr4: 00000048109F9620
2XHREGISTER      gpr5: 0000004808619800
...
2XHREGISTER      fpr13: 0000000000000000
2XHREGISTER      fpr14: 0000000000000000
2XHREGISTER      fpr15: 0000000000000000
2XHREGISTER      fpc: 0008000000000000
2XHREGISTER      psw0: 0785140180000000
2XHREGISTER      psw1: 0000000026FBF2B0
2XHREGISTER      sp: 00000048109F9620
2XHREGISTER      bea: 0000000026FBD758
NULL
...
```

Here is a sample on zLinux:

```
0SECTION       GPINFO subcomponent dump routine
NULL           ================================
2XHOSLEVEL     OS Level         : Linux 3.0.76-0.9-default
...
NULL
1XHREGISTERS   Registers:
2XHREGISTER      gpr0: 0000000000000000
2XHREGISTER      gpr1: 0000000000000000
2XHREGISTER      gpr2: 000003FFFD43DAD8
2XHREGISTER      gpr3: 000003FFFD43DAD8
2XHREGISTER      gpr4: 000003FFFD43DAD8
2XHREGISTER      gpr5: 0000000000000000
...
2XHREGISTER      fpr13: 0000000000000000
2XHREGISTER      fpr14: 0000000000000000
2XHREGISTER      fpr15: 0000000000000000
2XHREGISTER      psw: 000003FFFCBA8B70
2XHREGISTER      mask: 0705E00180000000
2XHREGISTER      fpc: 0008000000000000
2XHREGISTER      bea: 000003FFFCBA957C
NULL
...
```

The BEA register is useful for debugging wild branch problems, helping you to reconstruct the control flow paths that lead up to a crash.

ENVINFO

This section shows information about the runtime environment level that failed, the command used to start the JVM process, and the JVM environment. Additional information is provided that is included in the `1CIJAVAVERSION` line. This information is the final package build ID, shown in brackets at the end of the line.

The line, 1CIJITMODES, provides information about JIT settings. In earlier releases, some of the information about JIT and AOT settings is shown in the 1CIJITVERSION line.

The line 1CIPROCESSID shows the ID of the operating system process that produced the javacore.

```
NULL           ---------------------------------
0SECTION       ENVINFO subcomponent dump routine
NULL           =================================
1CIJAVAVERSION JRE 1.6.0 Windows 7 x86-32 build 20110928_91462 (pwi3260_26sr1-20110929_01(SR1))
1CIVMVERSION   VM build R26_JVM_26_20110927_1438_B91416
1CIJITVERSION  r11_20110916_20778
1CIGCVERSION   GC - R26_JVM_26_20110923_1426_B91192
1CIJITMODES    JIT enabled, AOT enabled, FSD disabled, HCR disabled
1CIRUNNINGAS   Running as a standalone JVM
1CIPROCESSID   Process ID: 14632 (0x3928)
1CICMDLINE     java -Xdump:java:events=vmstart -version
1CIJAVAHOMEDIR Java Home Dir: c:\build\pwi3260_26sr1-20110929\sdk\jre
1CIJAVADLLDIR  Java DLL Dir: c:\build\pwi3260_26sr1-20110929\sdk\jre\bin
1CISYSCP       Sys Classpath: c:\build\pwi3260_26sr1-20110929\sdk\jre\bin\default\jclSC160\vm.jar;....
1CIUSERARGS    UserArgs:
2CIUSERARG     -Xoptionsfile=c:\build\pwi3260_26sr1-20110929\sdk\jre\bin\default\options.default
```

On Linux platforms, the ENVINFO section contains additional information about the sched_compat_yield Linux kernel setting in force when the JVM was started. The typical output is:

```
1CISYSINFO     System Configuration
NULL           ------------------------
2CISYSINFO     /proc/sys/kernel/sched_compat_yield = 0
```

For further information about the effect of this kernel setting, see the details about the Linux Completely Fair Scheduler in Known limitations on Linux.

Native memory (NATIVEMEMINFO)

The NATIVEMEMINFO section of a Javadump provides information about the native memory allocated by the Java runtime environment).

Native memory is memory requested from the operating system using library functions such as malloc() and mmap().

When the runtime environment allocates native memory, the memory is associated with a high-level memory category. Each memory category has two running counters:

- The total number of bytes allocated but not yet freed.
- The number of native memory allocations that have not been freed.

Each memory category can have subcategories.

The NATIVEMEMINFO section provides a breakdown of memory categories by runtime environment component. Each memory category contains the total value for each counter in that category and all related subcategories.

The runtime environment tracks native memory allocated only by the Java runtime environment and class libraries. The runtime environment does not record memory allocated by application or third-party JNI code. The total native memory reported in the NATIVEMEMINFO section is always slightly less than the total native address space usage reported through operating system tools for the following reasons:

- The memory counter data might not be in a consistent state when the Javadump is taken.
- The data does not include any overhead introduced by the operating system.

A memory category for `Direct Byte Buffers` can be found in the `VM Class libraries` section of the `NATIVEMEMINFO` output.

```
0SECTION       NATIVEMEMINFO subcomponent dump routine
NULL           ================================
0MEMUSER
1MEMUSER       JRE: 591,281,600 bytes / 2763 allocations
1MEMUSER       |
2MEMUSER       +--VM: 575,829,048 bytes / 2143 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Classes: 14,357,408 bytes / 476 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Memory Manager (GC): 548,712,024 bytes / 435 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Java Heap: 536,870,912 bytes / 1 allocation
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Other: 11,841,112 bytes / 434 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Threads: 11,347,376 bytes / 307 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Java Stack: 378,832 bytes / 28 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Native Stack: 10,649,600 bytes / 30 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Other: 318,944 bytes / 249 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Trace: 324,464 bytes / 294 allocations
2MEMUSER       |  |
3MEMUSER       |  +--JVMTI: 17,784 bytes / 13 allocations
2MEMUSER       |  |
3MEMUSER       |  +--JNI: 129,760 bytes / 250 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Port Library: 10,240 bytes / 62 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Other: 929,992 bytes / 306 allocations
1MEMUSER       |
2MEMUSER       +--JIT: 14,278,744 bytes / 287 allocations
2MEMUSER       |  |
3MEMUSER       |  +--JIT Code Cache: 8,388,608 bytes / 4 allocations
2MEMUSER       |  |
3MEMUSER       |  +--JIT Data Cache: 2,097,216 bytes / 1 allocation
2MEMUSER       |  |
3MEMUSER       |  +--Other: 3,792,920 bytes / 282 allocations
1MEMUSER       |
2MEMUSER       +--Class Libraries: 1,173,808 bytes / 333 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Harmony Class Libraries: 2,000 bytes / 1 allocation
2MEMUSER       |  |
3MEMUSER       |  +--VM Class Libraries: 1,171,808 bytes / 332 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--sun.misc.Unsafe: 6,768 bytes / 5 allocations
4MEMUSER       |  |  |  |
5MEMUSER       |  |  |  +--Direct Byte Buffers: 6,120 bytes / 1 allocation
4MEMUSER       |  |  |  |
5MEMUSER       |  |  |  +--Other: 648 bytes / 4 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Other: 1,165,040 bytes / 327 allocations
NULL
NULL           ------------------------------------------------------------------------
```

Storage Management (MEMINFO)

The MEMINFO section provides information about the Memory Manager.

This section is different in Javadumps that are generated from an IBM J9 2.6 virtual machine. The information shows the free memory, used memory, and total memory for the heap, in decimal and hexadecimal values. If an initial maximum heap size, or *soft* limit, is specified using the **–Xsoftmx** option, this is also shown as the target memory for the heap. For more information about **–Xsoftmx**, see "-Xsoftmx" on page 168.

The MEMINFO section also contains garbage collection history data as a sequence of trace points, each with a timestamp, ordered with the most recent trace point first.

The following example shows some typical output. All of these values are generated as hexadecimal values. The column headings in the `MEMINFO` section have the following meanings:

- Object memory section (`HEAPTYPE`):

  id The ID of the space or region.

  start The start address of this region of the heap.

  end The end address of this region of the heap.

  size The size of this region of the heap.

  space/region
  For a line that contains only an `id` and a name, this column shows the name of the memory space. Otherwise the column shows the name of the memory space, followed by the name of a particular region that is contained within that memory space.

- Internal memory section (`SEGTYPE`), including class memory, JIT code cache, and JIT data cache:

  segment
  The address of the segment control data structure.

  start The start address of the native memory segment.

  alloc The current allocation address within the native memory segment.

  end The end address of the native memory segment.

  type An internal bit field describing the characteristics of the native memory segment.

  size The size of the native memory segment.

```
0SECTION       MEMINFO subcomponent dump routine
NULL           ================================
NULL
1STHEAPTYPE    Object Memory
NULL           id                 start              end                size               space/region
1STHEAPSPACE   0x000000000042D4B0         --                 --                 --         Generational
1STHEAPREGION  0x0000000000383C70 0x000007FFDFFB0000 0x000007FFE02B0000 0x0000000000300000 Generational/Tenured Region
1STHEAPREGION  0x0000000000383B80 0x000007FFFFEB0000 0x000007FFFFF30000 0x0000000000080000 Generational/Nursery Region
1STHEAPREGION  0x0000000000383A90 0x000007FFFFF30000 0x000007FFFFFB0000 0x0000000000080000 Generational/Nursery Region
NULL
1STHEAPTOTAL   Total memory:           4194304  (0x0000000000400000)
1STHEAPTARGET  Target memory:          20971520 (0x0000000001400000)
1STHEAPINUSE   Total memory in use:    1184528  (0x0000000000121310)
1STHEAPFREE    Total memory free:      3009776  (0x00000000002DECF0)
NULL
1STSEGTYPE     Internal Memory
NULL           segment            start              alloc              end                type       size
1STSEGMENT     0x0000000002CE3DF8 0x0000000003BD00F0 0x0000000003BD00F0 0x0000000003BE00F0 0x01000040 0x0000000000010000
1STSEGMENT     0x0000000002CE3D38 0x0000000003A509F0 0x0000000003A509F0 0x0000000003A609F0 0x01000040 0x0000000000010000
(lines removed for clarity)
1STSEGMENT     0x00000000004481D8 0x0000000002CE9B10 0x0000000002CE9B10 0x0000000002CF9B10 0x00800040 0x0000000000010000
NULL
```

```
1STSEGTOTAL    Total memory:           1091504 (0x000000000010A7B0)
1STSEGINUSE    Total memory in use:          0 (0x0000000000000000)
1STSEGFREE     Total memory free:      1091504 (0x000000000010A7B0)
NULL
1STSEGTYPE     Class Memory
NULL           segment            start              alloc              end                type       size
1STSEGMENT     0x0000000003B117B8 0x0000000003C4E210 0x0000000003C501C0 0x0000000003C6E210 0x00020040 0x0000000000020000
1STSEGMENT     0x0000000003B116F8 0x0000000003C451D0 0x0000000003C4D1D0 0x0000000003C4D1D0 0x00010040 0x0000000000008000
(lines removed for clarity)
1STSEGMENT     0x00000000004489E8 0x0000000003804A90 0x0000000003824120 0x0000000003824A90 0x00020040 0x0000000000020000
NULL
1STSEGTOTAL    Total memory:           2099868 (0x000000000200A9C)
1STSEGINUSE    Total memory in use:    1959236 (0x00000000001DE544)
1STSEGFREE     Total memory free:       140632 (0x0000000000022558)
NULL
1STSEGTYPE     JIT Code Cache
NULL           segment            start              alloc              end                type       size
1STSEGMENT     0x0000000002D5B508 0x000007FFDEE80000 0x000007FFDEEA2D78 0x000007FFDF080000 0x00000068 0x0000000000200000
1STSEGMENT     0x0000000002CE9688 0x000007FFDF080000 0x000007FFDF09FD58 0x000007FFDF280000 0x00000068 0x0000000000200000
1STSEGMENT     0x0000000002CE95C8 0x000007FFDF280000 0x000007FFDF29FD58 0x000007FFDF480000 0x00000068 0x0000000000200000
1STSEGMENT     0x0000000002CE9508 0x000007FFDF480000 0x000007FFDF49FD58 0x000007FFDF680000 0x00000068 0x0000000000200000
NULL
1STSEGTOTAL    Total memory:          8388608 (0x0000000000800000)
1STSEGINUSE    Total memory in use:    533888 (0x0000000000082580)
1STSEGFREE     Total memory free:     7854720 (0x000000000077DA80)
1STSEGLIMIT    Allocation limit:       268435456 (0x0000000010000000)
NULL
1STSEGTYPE     JIT Data Cache
NULL           segment            start              alloc              end                type       size
1STSEGMENT     0x0000000002CE9888 0x0000000003120060 0x0000000003121F58 0x0000000003320060 0x00000048 0x0000000000200000
NULL
1STSEGTOTAL    Total memory:          2097152 (0x0000000000200000)
1STSEGINUSE    Total memory in use:      7928 (0x0000000000001EF8)
1STSEGFREE     Total memory free:     2089224 (0x00000000001FE108)
1STSEGLIMIT    Allocation limit:       402653184 (0x0000000018000000)
NULL
1STGCHTYPE     GC History
3STHSTTYPE     14:54:17:123462116 GMT j9mm.134 -   Allocation failure end: newspace=111424/524288 oldspace=3010952/3145728
loa=156672/156672
3STHSTTYPE     14:54:17:123459726 GMT j9mm.470 -   Allocation failure cycle end: newspace=111448/524288 oldspace=
3010952/3145728 loa=156672/156672
3STHSTTYPE     14:54:17:123454948 GMT j9mm.65 -   LocalGC end: rememberedsetoverflow=0 causedrememberedsetoverflow=0
scancacheoverflow=0 failedflipcount=0 failedflipbytes=0 failedtenurecount=0 failedtenurebytes=0 flipcount=2561
flipbytes=366352 newspace=111448/524288 oldspace=3010952/3145728 loa=156672/156672 tenureage=10
3STHSTTYPE     14:54:17:123441638 GMT j9mm.140 -   Tilt ratio: 50
3STHSTTYPE     14:54:17:122664846 GMT j9mm.64 -   LocalGC start: globalcount=0 scavengecount=1 weakrefs=0 soft=0
phantom=0 finalizers=0
3STHSTTYPE     14:54:17:122655972 GMT j9mm.63 -   Set scavenger backout flag=false
3STHSTTYPE     14:54:17:122647781 GMT j9mm.135 -   Exclusive access: exclusiveaccessms=0.002 meanexclusiveaccessms=0.002
threads=0 lastthreadtid=0x0000000002DCCE00 beatenbyotherthread=0
3STHSTTYPE     14:54:17:122647440 GMT j9mm.469 -   Allocation failure cycle start: newspace=0/524288 oldspace=
3010952/3145728 loa=156672/156672 requestedbytes=24
3STHSTTYPE     14:54:17:122644709 GMT j9mm.133 -   Allocation failure start: newspace=0/524288 oldspace=3010952/3145728
loa=156672/156672 requestedbytes=24
NULL
```

Locks, monitors, and deadlocks (LOCKS)

An example of the LOCKS component part of a Javadump taken during a deadlock.

A lock typically prevents more than one entity from accessing a shared resource. Each object in the Java language has an associated lock, also referred to as a monitor, which a thread obtains by using a synchronized method or block of code. In the case of the JVM, threads compete for various resources in the JVM and locks on Java objects. In addition to locks that are obtained by using synchronized code, the Java language includes locks based on the `java.util.concurrent.locks` package.

When you take a Java dump, the JVM attempts to detect deadlock cycles. The JVM can detect cycles that consist of locks that are obtained through synchronization, locks that extend the `java.util.concurrent.locks.AbstractOwnableSynchronizer` class, or a mix of both lock types.

The following example is from a deadlock test program where two threads, "DeadLockThread 0" and "DeadLockThread 1", unsuccessfully attempt to synchronize on a java/lang/String object, and lock an instance of the `java.util.concurrent.locks.ReentrantLock` class.

The Locks section in the example (highlighted) shows that thread "DeadLockThread 1" locked the object instance java/lang/String@0x00007F5E5E18E3D8. The monitor was created as a result of a Java code fragment such as `synchronize(aString)`, and this monitor has "DeadLockThread 0" waiting to get a lock on this same object instance (`aString`). The deadlock section also shows an instance of the `java.util.concurrentlocks.ReentrantLock$NonfairSync` class, that is locked by "DeadLockThread 0", and has "Deadlock Thread 1" waiting.

This classic deadlock situation is caused by an error in application design; the Javadump tool is a major tool in the detection of such events.

Blocked thread information is also available in the Threads section of the Java dump, in lines that begin with 3XMTHREADBLOCK, for threads that are blocked, waiting or parked. For more information, see "Blocked thread information" on page 79.

```
NULL           ------------------------------------------------------------------------
0SECTION       LOCKS subcomponent dump routine
NULL           ================================
NULL
1LKPOOLINFO    Monitor pool info:
2LKPOOLTOTAL     Current total number of monitors: 2
NULL
1LKMONPOOLDUMP Monitor Pool Dump (flat & inflated object-monitors):
2LKMONINUSE      sys_mon_t:0x00007F5E24013F10 infl_mon_t: 0x00007F5E24013F88:
3LKMONOBJECT       java/lang/String@0x00007F5E5E18E3D8: Flat locked by "Deadlock Thread 1" (0x00007F5E84362100), entry count 1
3LKWAITERQ           Waiting to enter:
3LKWAITER              "Deadlock Thread 0" (0x00007F5E8435BD00)
NULL
1LKREGMONDUMP  JVM System Monitor Dump (registered monitors):
2LKREGMON          Thread global lock (0x00007F5E84004F58): <unowned>
2LKREGMON          &(PPG_mem_mem32_subAllocHeapMem32.monitor) lock (0x00007F5E84005000): <unowned>
2LKREGMON          NLS hash table lock (0x00007F5E840050A8): <unowned>
             < lines removed for brevity >

1LKDEADLOCK    Deadlock detected !!!
NULL           ---------------------
NULL
2LKDEADLOCKTHR  Thread "Deadlock Thread 0" (0x00007F5E8435BD00)
3LKDEADLOCKWTR    is waiting for:
4LKDEADLOCKMON      sys_mon_t:0x00007F5E24013F10 infl_mon_t: 0x00007F5E24013F88:
4LKDEADLOCKOBJ      java/lang/String@0x00007F5E5E18E3D8
3LKDEADLOCKOWN    which is owned by:
2LKDEADLOCKTHR  Thread "Deadlock Thread 1" (0x00007F5E84362100)
3LKDEADLOCKWTR    which is waiting for:
4LKDEADLOCKOBJ      java/util/concurrent/locks/ReentrantLock$NonfairSync@0x00007F5E7E1464F0
3LKDEADLOCKOWN    which is owned by:
2LKDEADLOCKTHR  Thread "Deadlock Thread 0" (0x00007F5E8435BD00)
```

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, native threads, and stack traces.

A Java thread is implemented by a native thread of the operating system. Each thread is represented by a set of lines such as:

```
3XMTHREADINFO "main" J9VMThread:0x002DA900, j9thread_t:0x00D84630, java/lang/Thread:0x227E0078, state:R,
prio=5
3XMJAVALTHREAD (java/lang/Thread getId:0x1, isDaemon:false)
3XMTHREADINFO1 (native thread ID:0xE28, native priority:0x5, native policy:UNKNOWN, vmstate:CW, vm thread flags:0x00000001)
3XMCPUTIME CPU usage total: 0.562500000 secs, user: 0.218750000 secs, system: 0.343750000 secs
3XMHEAPALLOC Heap bytes allocated since last GC cycle=36512 (0x8EA0)
3XMTHREADINFO3 Java callstack:
4XESTACKTRACE at java/lang/Thread.sleep(Native Method)
4XESTACKTRACE at java/lang/Thread.sleep(Thread.java:961)
```

```
4XESTACKTRACE at Sleep.main(Sleep.java:11)
3XMTHREADINFO3       Native callstack:
4XENATIVESTACK           ZwWaitForSingleObject+0xa (0x0000000077C612FA [ntdll+0x512fa])
4XENATIVESTACK           WaitForSingleObjectEx+0x9c (0x000007FEFDAB10DC [KERNELBASE+0x10dc])
4XENATIVESTACK           monitor_wait_original+0x83e (j9thread.c:3766, 0x000007FEF4C2600E [J9THR26+0x600e])
4XENATIVESTACK           j9thread_monitor_wait+0x43 (j9thread.c:3492, 0x000007FEF4C26993 [J9THR26+0x6993])
4XENATIVESTACK           internalAcquireVMAccessNoMutexWithMask+0x32c (vmaccess.c:320, 0x000007FEF1EDE02C [j9vm26+0x6e02c])
4XENATIVESTACK           javaCheckAsyncMessages+0xe9 (async.asm:156, 0x000007FEF1E81609 [j9vm26+0x11609])
```

The properties on the first line are the thread name, addresses of the JVM thread structures and of the Java thread object, thread state, and Java thread priority. For Java threads, the second line contains the thread ID and daemon status from the Java thread object.The next line includes the following properties:

- Native operating system thread ID
- Native operating system thread priority
- Native operating system scheduling policy
- Internal VM thread state
- Internal VM thread flags

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.

If a thread is parked or blocked, the output contains a line for that thread, beginning with `3XMTHREADBLOCK`, listing the resource that the thread is waiting for and, if possible, the thread that currently owns that resource. For more information see “Blocked thread information” on page 79.

For Java threads and attached native threads, the output contains a line beginning with `3XMCPUTIME`, which displays the number of seconds of CPU time that was consumed by the thread since that thread was started. The total CPU time that is consumed by a thread is reported. On AIX and Windows, the time that is consumed in user code and in system code is also reported. If a Java thread is re-used from a thread pool, the CPU counts for that thread are not reset, and continue to accumulate.

For Java threads, the line beginning with `3XMHEAPALLOC` displays the number of bytes of Java objects and arrays allocated by that thread since the last garbage collection cycle. In the example, this line is just before the Java callstack.

If the Java dump was triggered by an exception throw, catch, uncaught, or systhrow event, or by the com.ibm.jvm.Dump API, the output contains the stored tracepoint history for the thread. For more information, see "Trace history for the current thread" on page 82.

When you initiate a javadump to obtain diagnostic information, the JVM quiesces Java threads before producing the javacore. A preparation state of `exclusive_vm_access` is shown in the `1TIPREPSTATE` line of the `TITLE` section.

```
1TIPREPSTATE Prep State: 0x4 (exclusive_vm_access)
```

Threads that were running Java code when the javacore was triggered have a Java thread state of R (Runnable) and an internal VM thread state of CW (Condition Wait).

Previous behavior

Before service refresh 8, fix pack 4, threads that were running Java code when the javacore was triggered show the thread state as in CW (Condition Wait) state, for example:

```
3XMTHREADINFO "main" J9VMThread:0x002DA900, j9thread_t:0x00D84630, java/lang/Thread:0x227E0078, state:CW,
prio=5
3XMJAVALTHREAD (java/lang/Thread getId:0x1, isDaemon:false)
3XMTHREADINFO1 (native thread ID:0xE28, native priority:0x5, native policy:UNKNOWN)
```

The javacore `LOCKS` section shows that these threads are waiting on an internal JVM lock.

```
2LKREGMON          Thread public flags mutex lock (0x002A5234): <unowned>
3LKNOTIFYQ            Waiting to be notified:
3LKWAITNOTIFY            "main" (0x002DA900)
```

Understanding Java and native thread details:

After each thread heading are the stack traces, which can be separated into three types; Java threads, attached native threads and unattached native threads. There is some extra information shown when the IBM SDK, Java Technology Edition, Version 6 uses the IBM J9 2.6 virtual machine.

The following examples are taken from 32-bit Windows. Other platforms provide different levels of detail for the native stack.

Java thread

A Java thread runs on a native thread, which means that there are two stack traces for each Java thread. The first stack trace shows the Java methods and the second stack trace shows the native functions. This example is an internal Java thread:

```
3XMTHREADINFO "Attach API wait loop" J9VMThread:0x23783D00, j9thread_t:0x026958F8, java/lang/Thread:0x027F0640, state:R, prio=10
3XMJAVALTHREAD (java/lang/Thread getId:0xB, isDaemon:true)
3XMTHREADINFO1 (native thread ID:0x15C, native priority:0xA, native policy:UNKNOWN)
3XMCPUTIME CPU usage total: 0.562500000 secs, user: 0.218750000 secs, system: 0.343750000 secs
3XMHEAPALLOC Heap bytes allocated since last GC cycle=0 (0x0)
3XMTHREADINFO3 Java callstack:
4XESTACKTRACE at com/ibm/tools/attach/javaSE/IPC.waitSemaphore(Native Method)
4XESTACKTRACE at com/ibm/tools/attach/javaSE/CommonDirectory.waitSemaphore(CommonDirectory.java:193)
4XESTACKTRACE at com/ibm/tools/attach/javaSE/AttachHandler$WaitLoop.waitForNotification(AttachHandler.java:337)
4XESTACKTRACE at com/ibm/tools/attach/javaSE/AttachHandler$WaitLoop.run(AttachHandler.java:415)
3XMTHREADINFO3 Native callstack:
4XENATIVESTACK ZwWaitForSingleObject+0x15 (0x7787F8B1 [ntdll+0x1f8b1])
4XENATIVESTACK WaitForSingleObjectEx+0x43 (0x75E11194 [kernel32+0x11194])
4XENATIVESTACK WaitForSingleObject+0x12 (0x75E11148 [kernel32+0x11148])
4XENATIVESTACK j9shsem_wait+0x94 (j9shsem.c:233, 0x7460C394 [J9PRT26+0xc394])
4XENATIVESTACK Java_com_ibm_tools_attach_javaSE_IPC_waitSemaphore+0x48 (attach.c:480, 0x6FA61E58 [jclse7b_26+0x1e58])
4XENATIVESTACK VMprJavaSendNative+0x504 (jnisend.asm:521, 0x709746D4 [j9vm26+0x246d4])
4XENATIVESTACK javaProtectedThreadProc+0x9d (vmthread.c:1868, 0x709A05BD [j9vm26+0x505bd])
4XENATIVESTACK j9sig_protect+0x44 (j9signal.c:150, 0x7460F0A4 [J9PRT26+0xf0a4])
```

```
4XENATIVESTACK javaThreadProc+0x39 (vmthread.c:298, 0x709A0F39 [j9vm26+0x50f39])
4XENATIVESTACK thread_wrapper+0xda (j9thread.c:1234, 0x7497464A [J9THR26+0x464a])
4XENATIVESTACK _endthread+0x48 (0x7454C55C [msvcr100+0x5c55c])
4XENATIVESTACK _endthread+0xe8 (0x7454C5FC [msvcr100+0x5c5fc])
4XENATIVESTACK BaseThreadInitThunk+0x12 (0x75E1339A [kernel32+0x1339a])
4XENATIVESTACK RtlInitializeExceptionChain+0x63 (0x77899EF2 [ntdll+0x39ef2])
4XENATIVESTACK RtlInitializeExceptionChain+0x36 (0x77899EC5 [ntdll+0x39ec5])
```

The Java stack trace includes information about locks that were taken within that stack by calls to synchronized methods or the use of the `synchronized` keyword.

After each stack frame in which one or more locks were taken, the Java stack trace might include extra lines starting with 5XESTACKTRACE. These lines show the locks that were taken in the method on the previous line in the trace, and a cumulative total of how many times the locks were taken within that stack at that point. This information is useful for determining the locks that are held by a thread, and when those locks will be released.

Java locks are re-entrant; they can be entered more than once. Multiple occurrences of the `synchronized` keyword in a method might result in the same lock being entered more than once in that method. Because of this behavior, the entry counts might increase by more than one, between two method calls in the Java stack, and a lock might be entered at multiple positions in the stack. The lock is not released until the first entry, the one furthest down the stack, is released.

Java locks are released when the `Object.wait()` method is called. Therefore a record of a thread entering a lock in its stack does not guarantee that the thread still holds the lock. The thread might be waiting to be notified about the lock, or it might be blocked while attempting to re-enter the lock after being notified. In particular, if another thread calls the `Object.notifyAll()` method, all threads that are waiting for that monitor must compete to re-enter it, and some threads will become blocked. You can determine whether a thread is blocked or waiting on a lock by looking at the 3XMTHREADBLOCK line for that thread. For more information see "Blocked thread information." A thread that calls the `Object.wait()` method releases the lock only for the object that it called the `Object.wait()` method on. All other locks that the thread entered are still held by that thread.

The following lines show an example Java stack trace for a thread that calls `java.io.PrintStream` methods:

```
4XESTACKTRACE at java/io/PrintStream.write(PrintStream.java:504(Compiled Code))
5XESTACKTRACE (entered lock: java/io/PrintStream@0xA1960698, entry count: 3)
4XESTACKTRACE at sun/nio/cs/StreamEncoder.writeBytes(StreamEncoder.java:233(Compiled Code))
4XESTACKTRACE at sun/nio/cs/StreamEncoder.implFlushBuffer(StreamEncoder.java:303(Compiled Code))
4XESTACKTRACE at sun/nio/cs/StreamEncoder.flushBuffer(StreamEncoder.java:116(Compiled Code))
5XESTACKTRACE (entered lock: java/io/OutputStreamWriter@0xA19612D8, entry count: 1)
4XESTACKTRACE at java/io/OutputStreamWriter.flushBuffer(OutputStreamWriter.java:203(Compiled Code))
4XESTACKTRACE at java/io/PrintStream.write(PrintStream.java:551(Compiled Code))
5XESTACKTRACE (entered lock: java/io/PrintStream@0xA1960698, entry count: 2)
4XESTACKTRACE at java/io/PrintStream.print(PrintStream.java:693(Compiled Code))
4XESTACKTRACE at java/io/PrintStream.println(PrintStream.java:830(Compiled Code))
5XESTACKTRACE (entered lock: java/io/PrintStream@0xA1960698, entry count: 1)
```

Blocked thread information:

For threads that are in parked, waiting, or blocked states, the Javadump `THREADS` section contains information about the resource that the thread is waiting for. The information might also include the thread that currently owns that resource. Use this information to solve problems with blocked threads.

Information about the state of a thread can be found in the THREADS section of the Javadump output. Look for the line that begins with 3XMTHREADINFO. The following states apply:

state:P
Parked threads

state:B
Blocked threads

state:CW
Waiting threads

To find out which resource is holding the thread in parked, waiting, or blocked state, look for the line that begins 3XMTHREADBLOCK. This line might also indicate which thread owns that resource.

The 3XMTHREADBLOCK section is not produced for threads that are blocked or waiting on a JVM System Monitor, or threads that are in Thread.sleep().

Threads enter the parked state through the java.util.concurrent API. Threads enter the blocked state through the Java synchronization operations.

The locks that are used by blocked and waiting threads are shown in the LOCKS section of the Javadump output, along with the thread that is owning the resource and causing the block. Locks that are being waited on might not have an owner. The waiting thread remains in waiting state until it is notified, or until the timeout expires. Where a thread is waiting on an unowned lock the lock is shown as Owned by: <unowned>.

Parked threads are listed as parked on the *blocker* object that was passed to the underlying java.util.concurrent.locks.LockSupport.park() method, if such an object was supplied. If a blocker object was not supplied, threads are listed as Parked on: <unknown>.

If the object that was passed to the park() method extends the java.util.concurrent.locks.AbstractOwnableSynchronizer class, and uses the methods of that class to keep track of the owning thread, then information about the owning thread is shown. If the object does not use the AbstractOwnableSynchronizer class, the owning thread is listed as <unknown>. The AbstractOwnableSynchronizer class is used to provide diagnostic data, and is extended by other classes in the java.util.concurrent.locks package. If you develop custom locking code with the java.util.concurrent package then you can extend and use the AbstractOwnableSynchronizer class to provide information in Java dumps to help you diagnose problems with your locking code.

Example: a blocked thread

The following sample output from the THREADS section of a Javadump shows a thread, Thread-5, that is in the blocked state, state:B. The thread is waiting for the resource java/lang/String@0x4D8C90F8, which is currently owned by thread main.

```
3XMTHREADINFO      "Thread-5" J9VMThread:0x4F6E4100, j9thread_t:0x501C0A28, java/lang/Thread:0x4D8C9520,
state:B, prio=5
3XMTHREADINFO1            (native thread ID:0x664, native priority:0x5, native policy:UNKNOWN)
3XMTHREADBLOCK     Blocked on: java/lang/String@0x4D8C90F8 Owned by: "main" (J9VMThread:0x00129100, java/
lang/Thread:0x00DD4798)
```

The LOCKS section of the Javadump shows the following, corresponding output about the block:

```
1LKMONPOOLDUMP Monitor Pool Dump (flat & inflated object-monitors):
2LKMONINUSE      sys_mon_t:0x501C18A8 infl_mon_t: 0x501C18E4:
3LKMONOBJECT       java/lang/String@0x4D8C90F8: Flat locked by "main" (0x00129100), entry count 1
3LKWAITERQ           Waiting to enter:
3LKWAITER               "Thread-5" (0x4F6E4100)
```

Look for information about the blocking thread, `main`, elsewhere in the `THREADS` section of the Javadump, to understand what that thread was doing when the Javadump was taken.

Example: a waiting thread

The following sample output from the `THREADS` section of a Javadump shows a thread, `Thread-5`, that is in the waiting state, `state:CW`. The thread is waiting to be notified on `java/lang/String@0x68E63E60`, which is currently owned by thread `main`:

```
3XMTHREADINFO      "Thread-5" J9VMThread:0x00503D00, j9thread_t:0x00AE45C8, java/lang/Thread:0x68E04F90,
state:CW, prio=5
3XMTHREADINFO1            (native thread ID:0xC0C, native priority:0x5, native policy:UNKNOWN)
3XMTHREADBLOCK     Waiting on: java/
lang/String@0x68E63E60 Owned by: "main" (J9VMThread:0x6B3F9A00, java/lang/Thread:0x68E64178)
```

The LOCKS section of the Javadump shows the corresponding output about the monitor being waited on:

```
1LKMONPOOLDUMP Monitor Pool Dump (flat & inflated object-monitors):
2LKMONINUSE      sys_mon_t:0x00A0ADB8 infl_mon_t: 0x00A0ADF4:
3LKMONOBJECT       java/lang/String@0x68E63E60: owner "main" (0x6B3F9A00), entry count 1
3LKNOTIFYQ           Waiting to be notified:
3LKWAITNOTIFY           "Thread-5" (0x00503D00)
```

Example: a parked thread that uses the AbstractOwnableSynchronizer class

The following sample output shows a thread, `Thread-5`, in the parked state, `state:P`. The thread is waiting to enter a java.util.concurrent.locks.ReentrantLock lock that uses the AbstractOwnableSynchronizer class:

```
3XMTHREADINFO      "Thread-5" J9VMThread:0x4F970200, j9thread_t:0x501C0A28, java/lang/Thread:0x4D9AD640,
state:P, prio=5
3XMTHREADINFO1            (native thread ID:0x157C, native priority:0x5, native policy:UNKNOWN)
3XMTHREADBLOCK     Parked on: java/util/concurrent/locks/ReentrantLock$NonfairSync@0x4D9ACCF0 Owned by:
"main" (J9VMThread:0x00129100, java/lang/Thread:0x4D943CA8)
```

This example shows both the reference to the `J9VMThread` thread and the `java/lang/Thread` thread that currently own the lock. However in some cases the J9VMThread thread is null:

```
3XMTHREADINFO      "Thread-6" J9VMThread:0x4F608D00, j9thread_t:0x501C0A28, java/lang/Thread:0x4D92AE78,
state:P, prio=5
3XMTHREADINFO1            (native thread ID:0x8E4, native priority:0x5, native policy:UNKNOWN)
3XMTHREADBLOCK     Parked on: java/util/concurrent/locks/ReentrantLock$FairSync@0x4D92A960 Owned by:
"Thread-5" (J9VMThread: <null>, java/lang/Thread:0x4D92AA58)
```

In this example, the thread that is holding the lock, `Thread-5`, ended without using the unlock() method to release the lock. Thread `Thread-6` is now deadlocked. The `THREADS` section of the Javadump will not contain another thread with a `java/lang/Thread` reference of 0x4D92AA58. (The name `Thread-5` could be reused by another thread, because there is no requirement for threads to have unique names.)

Example: a parked thread that is waiting to enter a user-written lock that does not use the AbstractOwnableSynchronizer class

Because the lock does not use the AbstractOwnableSynchronizer class, no information is known about the thread that owns the resource:

```
3XMTHREADINFO      "Thread-5" J9VMThread:0x4FBA5400, j9thread_t:0x501C0A28, java/lang/Thread:0x4D918570,
state:P, prio=5
3XMTHREADINFO1            (native thread ID:0x1A8, native priority:0x5, native policy:UNKNOWN)
3XMTHREADBLOCK     Parked on: SimpleLock@0x4D917798 Owned by: <unknown>
```

Example: a parked thread that called the LockSupport.park method without supplying a blocker object

Because a blocker object was not passed to the park() method, no information is known about the locked resource:

```
3XMTHREADINFO      "Thread-5" J9VMThread:0x4F993300, j9thread_t:0x501C0A28, java/lang/Thread:0x4D849028,
state:P, prio=5
3XMTHREADINFO1            (native thread ID:0x1534, native priority:0x5, native policy:UNKNOWN)
3XMTHREADBLOCK     Parked on: <unknown> Owned by: <unknown>
```

The last two examples provide little or no information about the cause of the block. If you want more information, you can write your own locking code by following the guidelines in the API documentation for the java.util.concurrent.locks.LockSupport and java.util.concurrent.locks.AbstractOwnableSynchronizer classes. By using these classes, your locks can provide details to monitoring and diagnostic tools, which helps you to determine which threads are waiting and which threads are holding locks.

Trace history for the current thread:

Some Java dumps show recent trace history for the current thread. You can use this information to diagnose the cause of Java exceptions.

For Java dumps that were triggered by exception throw, catch, uncaught, and systhrow events (see "Dump events" on page 66 for more information) or by the com.ibm.jvm.Dump API, extra lines are output at the end of the THREADS section. These lines show the stored tracepoint history for the thread, with the most recent tracepoints first. The trace data is introduced by the following line:

```
1XECTHTYPE     Current thread history (J9VMThread:0x0000000002BA0500)
```

The tracepoints provide detailed information about the JVM, JIT, or class library code that was run on the thread immediately before the Java dump was triggered. In the following example, the Java dump was triggered by a java/lang/VerifyError exception. The tracepoints show that the reason for the exception was that a method in a class was overridden, but is defined as `final` in a superclass (see tracepoint j9vm.91 in the example). The output also shows the names of the classes that were being loaded by the JVM when the exception occurred.

```
3XEHSTTYPE (time) j9dmp.9 - Preparing for dump, filename=C:\test\verifyerror\javacore.20140124.155651.4544.0001.txt
3XEHSTTYPE (time) j9vm.2 -  <Created RAM class 0000000000000000 from ROM class 00000000105E9DD0
3XEHSTTYPE (time) j9vm.304 - >setCurrentExceptionUTF
3XEHSTTYPE (time) j9vm.301 - <setCurrentException
3XEHSTTYPE (time) j9vm.5 -  <exitInterpreter
3XEHSTTYPE (time) j9vm.10 - >internalSendExceptionConstructor
3XEHSTTYPE (time) j9vm.353 - <loader 0000000002CC1900 class 0X000000000244DFA8 attemptDynamicClassLoad exit
3XEHSTTYPE (time) j9vm.2 -  <Created RAM class 0000000002CC1900 from ROM class 00000000105D27E0
3XEHSTTYPE (time) j9vm.80 -  ROM class 00000000105D27E0 is named java/lang/VerifyError
3XEHSTTYPE (time) j9vm.1 -  >Create RAM class from ROM class 00000000105D27E0 in class loader 000000000F4CE448
3XEHSTTYPE (time) j9vm.351 - >loader 00000000FFF70000 class java/lang/VerifyError attemptDynamicClassLoad entry
3XEHSTTYPE (time) j9vm.248 - <dispatchAsyncEvents
3XEHSTTYPE (time) j9vm.247 - call event handler: handlerKey=0 eventHandler=000007FEED7EAF20 userData=0000000000000000
```

```
3XEHSTTYPE (time) j9vm.246 - >dispatchAsyncEvents asyncEventFlags=0000000000000001
3XEHSTTYPE (time) j9vm.119 - send loadClass(java/lang/VerifyError), stringObject: 00000000FFFE85C0 loader: 00000000FFF70000
3XEHSTTYPE (time) j9vm.316 - 0X0000000002BA0500 loader 000000000F4CE448 class java/lang/VerifyError arbitratedLoadClass calling callLoadClass
3XEHSTTYPE (time) j9vm.318 - 0X0000000002BA0500 loader 000000000F4CE448 contendedLoadTableAddThread className java/lang/VerifyError count 1
3XEHSTTYPE (time) j9vm.315 - 0X0000000002BA0500 loader 000000000F4CE448 class java/lang/VerifyError arbitratedLoadClass enter className
3XEHSTTYPE (time) j9vm.294 - >setCurrentException index=55 constructorIndex=0 detailMessage=00000000FFFE8528
3XEHSTTYPE (time) j9vm.302 - >setCurrentExceptionUTF exceptionNumber=55 detailUTF=JVMVRFY007 final method overridden; class=B, method=bad()V
3XEHSTTYPE (time) j9vm.91 - * Method bad.()V is overridden, but it is final in a superclass. Throw VerifyError
3XEHSTTYPE (time) j9vm.319 - 0X0000000002BA0500 loader 000000000F4CE5F8 class A arbitratedLoadClass exit foundClass 0000000002CC9600
3XEHSTTYPE (time) j9vm.120 - sent loadClass(A) -- got 00000000E0010D60
3XEHSTTYPE (time) j9vm.2 -  <Created RAM class 0000000002CC9600 from ROM class 00000000105E9FB0
3XEHSTTYPE (time) j9vm.80 -  ROM class 00000000105E9FB0 is named A
3XEHSTTYPE (time) j9vm.1 -  >Create RAM class from ROM class 00000000105E9FB0 in class loader 000000000F4CE5F8
3XEHSTTYPE (time) j9bcverify.18 - j9bcv_verifyClassStructure
3XEHSTTYPE (time) j9bcverify.14 - j9bcv_verifyClassStructure - class: A
3XEHSTTYPE (time) j9vm.353 - <loader 0000000000000000 class 0X000000000244E5F8 attemptDynamicClassLoad exit
3XEHSTTYPE (time) j9vm.351 - >loader 00000000FFF70000 class A attemptDynamicClassLoad entry
3XEHSTTYPE (time) j9vm.119 - send loadClass(A), stringObject: 00000000FFFE7DB8 loader: 00000000FFF70110
3XEHSTTYPE (time) j9vm.315 - >0X0000000002BA0500 loader 000000000F4CE5F8 class A arbitratedLoadClass enter className
3XEHSTTYPE (time) j9vm.80 -  ROM class 00000000105E9DD0 is named B
3XEHSTTYPE (time) j9vm.1 -   >Create RAM class from ROM class 00000000105E9DD0 in class loader 000000000F4CE5F8
3XEHSTTYPE (time) j9bcverify.18 - j9bcv_verifyClassStructure
3XEHSTTYPE (time) j9bcverify.14 - j9bcv_verifyClassStructure - class: B
```

Shared Classes (SHARED CLASSES)

An example of the shared classes section that includes summary information about the shared data cache.

See "printStats utility" on page 106 for a description of the summary information.

In service refresh 1, information is provided for the reserved and maximum space for AOT and JIT data bytes.

```
------------------------------------------------------------------------
SHARED CLASSES subcomponent dump routine
========================================

Cache Created With
------------------

-Xnolinenumbers = false

Cache Summary
-------------------

No line number content = false
Line number content = true

ROMClass start address = 0x629EC000
ROMClass end address = 0x62AD1468
Metadata start address = 0x636F9800
Cache end address = 0x639D0000
Runtime flags = 0x00000001ECA6029F
Cache generation = 13

Cache size = 16776768
Free bytes = 12747672
ROMClass bytes = 939112
AOT code bytes = 0
AOT data bytes = 0
AOT class hierarchy bytes = 0
AOT thunk bytes = 0
Reserved space for AOT bytes = -1
Maximum space for AOT bytes = -1
JIT hint bytes = 0
JIT profile bytes = 2280
Reserved space for JIT data bytes = -1
Maximum space for JIT data bytes = -1
Java Object bytes = 0
Zip cache bytes = 791856
ReadWrite bytes = 114240
JCL data bytes = 0
```

```
Byte data bytes = 0
Metadata bytes = 18920
Class debug area size = 2162688
Class debug area % used = 7%
Class LineNumberTable bytes = 97372
Class LocalVariableTable bytes = 57956

Number ROMClasses = 370
Number AOT Methods = 0
Number AOT Data Entries = 0
Number AOT Class Hierarchy = 0
Number AOT Thunks = 0
Number JIT Hints = 0
Number JIT Profiles = 24
Number Classpaths = 1
Number URLs = 0
Number Tokens = 0
Number Java Objects = 0
Number Zip Caches = 28
Number JCL Entries = 0
Number Stale classes = 0
Percent Stale classes = 0%

Cache is 12% full

Cache Memory Status
------------------
Cache Name Memory type Cache path

sharedcc_tempcache Memory mapped file C:\Documents and Settings\Administrator\Local
Settings\Application Data\javasharedresources\C260M2A32P_sharedcc_tempcache_G13

Cache Lock Status
------------------
Lock Name Lock type TID owning lock

Cache write lock File lock Unowned
Cache read/write lock File lock Unowned
```

Using Heapdump

The term Heapdump describes the IBM JVM mechanism that generates a dump of all the live objects that are on the Java heap; that is, objects that are being used by the running Java application.

Heapdump generates files that contain a list of objects that are in the Java heap. There are two supported Heapdump formats

- Portable Heap Dump (PHD) format
- text or classic format

The content of PHD Heapdumps has changed. Instead of 16-bit hashcodes, the IBM J9 2.6 virtual machine now has 32-bit hashcodes.

The content and range of information in a Heapdump might change between JVM versions or service refreshes.

Portable Heap Dump (PHD) file format

A PHD Heapdump file contains a header, plus a number of records that describe objects, arrays, and classes.

This description of the PHD Heapdump file format includes references to primitive numbers, which are listed here with lengths:

- “byte”: 1 byte in length.

- "short": 2 byte in length.
- "int": 4 byte in length.
- "long": 8 byte in length.
- "word": 4 bytes in length on 32-bit platforms, or 8 bytes on 64-bit platforms.

The general structure of a PHD file consists of these elements:

- The UTF string "portable heap dump".
- An "int" containing the PHD version number.
- An "int" containing flags:
  - A value of 1 indicates that the "word" length is 64-bit.
  - A value of 2 indicates that all the objects in the dump are hashed. This flag is set for Heapdumps that use 16-bit hashcodes, that is, IBM SDK, JavaTechnology Edition 5.0 or 6 with an IBM J9 2.3, 2.4, or 2.5 virtual machine (VM). This flag is not set for IBM SDK, JavaTechnology Edition 6 when the product includes the IBM J9 2.6 virtual machine. These Heapdumps use 32-bit hashcodes that are only created when used. For example, these hashcodes are created when the APIs Object.hashCode() or Object.toString() are called in a Java application. If this flag is not set, the presence of a hashcode is indicated by the hashcode flag on the individual PHD records.
  - A value of 4 indicates that the dump is from an IBM J9 VM.
- A "byte" containing a tag that indicates the start of the header. The tag value is 1.
- A number of header records. These records are preceded by a one-byte header tag. The header record tags have a different range of values from the body, or object record tags. The end of the header is indicated by the end of header tag. Header records are optional.
  - header tag 1. Not used in Heapdumps generated by the IBM J9 VM.
  - header tag 2. Indicates the end of the header.
  - header tag 3. Not used in Heapdumps generated by the IBM J9 VM.
  - header tag 4. This tag is a UTF string that indicates the JVM version. The string has a variable length.
- A "byte" containing the "start of dump" body tag, with a tag value of 2.
- A number of dump records. These records are preceded by a 1 byte tag. The possible record types are:
  - Short object record. Indicated by having the 0x80 bit of the tag set.
  - Medium object record. Indicated by having the 0x40 bit of the tag set, and the top bit with a value of 0.
  - Primitive array record. Indicated by having the 0x20 bit of the tag set. All other tag values have the top 3 bits with a value of 0.
  - Long object record. Indicated by having a tag value of 4.
  - Object array record. Indicated by having a tag value of 5.
  - Class record. Indicated by having a tag value of 6.
  - Long primitive array record. Indicated by having a tag value of 7.
  - Object array record (revised). Indicated by having a tag value of 8.

  See later sections for more information about these record types.
- A "byte" containing a tag that indicates the end of the Heapdump. This tag has a value of 3.

The current PHD version is 5, which can be found in the following releases:

- IBM SDK, JavaTechnology Edition 5.0 service refresh 9 and later (APAR IZ34218)
- IBM SDK, JavaTechnology Edition 6 with one of the following IBM J9 virtual machine levels 2.4, 2.5, or 2.6.

PHD version 4 is found in IBM SDK, JavaTechnology Edition 5.0 service refresh 8 and earlier. These versions use an IBM J9 2.3 virtual machine.

PHD object records

PHD files can contain short, medium, and long object records, depending on the number of object references in the Heapdump.

Short object record

The short object record includes detailed information within the tag "byte". This information includes:

- The 1 byte tag. The top bit (0x80) is set and the following 7 bits in descending order contain:
  - 2 bits for the class cache index. The value represents an index into a cache of the last four classes used.
  - 2 bits containing the number of references. Most objects contain 0 - 3 references. If there are 4 - 7 references, the medium object record is used. If there are more than seven references, the long object record is used.
  - 1 bit to indicate whether the gap is a "byte" or a "short". The gap is the difference between the address of this object and the previous object. If set, the gap is a "short". If the gap does not fit into a "short", the "long" object record form is used.
  - 2 bits indicating the size of each reference. The following values apply:
    - `0` indicates "byte" format.
    - `1` indicates "short" format.
    - `2` indicates "integer" format.
    - `3` indicates "long" format.
- A "byte" or a "short" containing the gap between the address of this object and the address of the preceding object. The value is signed and represents the number of 32-bit "words" between the two addresses. Most gaps fit into 1 byte.
- If all objects are hashed, a "short" containing the hashcode.
- The array of references, if references exist. The tag shows the number of elements, and the size of each element. The value in each element is the gap between the address of the references and the address of the current object. The value is a signed number of 32-bit "words". Null references are not included.

Medium object record

These records provide the actual address of the class rather than a cache index. The format is:

- The 1 byte tag. The second bit (0x40) is set and the following 6 bits in descending order contain:
  - 3 bits containing the number of references.
  - 1 bit to indicate whether the gap is a 1 byte value or a "short" For more information, see the description in the short record format.
  - 2 bits indicating the size of each reference. For more information, see the description in the short record format.

- A “byte” or a “short” containing the gap between the address of this object and the address of the preceding object. For more information, see the description in the short record format.
- A “word” containing the address of the class of this object.
- If all objects are hashed, a “short” containing the hashcode.
- The array of references. For more information, see the description in the short record format.

Long object record

This record format is used when there are more than seven references, or if there are extra flags or a hashcode. The record format is:

- The 1 byte tag, containing the value 4.
- A “byte” containing flags, with these bits in descending order:
  - 2 bits to indicate whether the gap is a “byte”, “short”, “int” or “long” format.
  - 2 bits indicating the size of each reference. For more information, see the description in the short record format.
  - 2 unused bits.
  - 1 bit indicating if the object was hashed and moved. If this bit is set then the record includes the hashcode.
  - 1 bit indicating if the object was hashed.
- A “byte”, “short”, “int” or “long” containing the gap between the address of this object and the address of the preceding object. For more information, see the description in the short record format.
- A “word” containing the address of the class of this object.
- If all objects are hashed, a “short” containing the hashcode. Otherwise, an optional “int” containing the hashcode if the hashed and moved bit is set in the record flag byte.
- An “int” containing the length of the array of references.
- The array of references. For more information, see the description in the short record format.

PHD array records

PHD array records can cover primitive arrays and object arrays.

Primitive array record

The primitive array record contains:

- The 1 byte tag. The third bit (0x20) is set and the following 5 bits in descending order contain:
  - 3 bits containing the array type. The array type values are:
    - 0 = bool
    - 1 = char
    - 2 = float
    - 3 = double
    - 4 = byte
    - 5 = short
    - 6 = int
    - 7 = long

  - 2 bits indicating the length of the array size and also the length of the gap. These values apply:
    - 0 indicates a “byte”.
    - 1 indicates a “short”.
    - 2 indicates an “int”.
    - 3 indicates a “long”.
- “byte”, “short”, “int” or “long” containing the gap between the address of this object and the address of the preceding object. For more information, see the description in the short object record format.
- “byte”, “short”, “int” or “long” containing the array length.
- If all objects are hashed, a “short” containing the hashcode.

Long primitive array record

The long primitive array record is used when a primitive array has been hashed. The format is:

- The 1 byte tag containing the value 7.
- A “byte” containing flags, with these bits in descending order:
  - 3 bits containing the array type. For more information, see the description of the primitive array record.
  - 1 bit indicating the length of the array size and also the length of the gap. The range for this value includes:
    - 0 indicating a “byte”.
    - 1 indicating a “word”.
  - 2 unused bits.
  - 1 bit indicating if the object was hashed and moved. If this bit is set, the record includes the hashcode.
  - 1 bit indicating if the object was hashed.
- a “byte” or “word” containing the gap between the address of this object and the address of the preceding object. For more information, see the description in the short object record format.
- a “byte” or “word” containing the array length.
- If all objects are hashed, a “short” containing the hashcode. Otherwise, an optional “int” containing the hashcode if the hashed and moved bit is set in the record flag byte.

Object array record

The object array record format is:

- The 1 byte tag containing the value 5.
- A “byte” containing flags with these bits in descending order:
  - 2 bits to indicate whether the gap is “byte”, “short”, “int” or “long”.
  - 2 bits indicating the size of each reference. For more information, see the description in the short record format.
  - 2 unused bits.
  - 1 bit indicating if the object was hashed and moved. If this bit is set, the record includes the hashcode.
  - 1 bit indicating if the object was hashed.

- A “byte”, “short”, “int” or “long” containing the gap between the address of this object and the address of the preceding object. For more information, see the description in the short record format.
- A “word” containing the address of the class of the objects in the array. Object array records do not update the class cache.
- If all objects are hashed, a “short” containing the hashcode. If the hashed and moved bit is set in the records flag, this field contains an “int”.
- An “int” containing the length of the array of references.
- The array of references. For more information, see the description in the short record format.

Object array record (revised) - from PHD version 5

This array record is similar to the previous array record with two key differences:

1. The tag value is 8.
2. An extra “int” value is shown at the end. This int contains the true array length, shown as a number of array elements. The true array length might differ from the length of the array of references because null references are excluded.

This record type was added in PHD version 5.

PHD class records

The PHD class record encodes a class object.

Class record

The format of a class record is:

- The 1 byte tag, containing the value 6.
- A “byte” containing flags, with these bits in descending order:
  - 2 bits to indicate whether the gap is a “byte”, “short”, “int” or “long”.
  - 2 bits indicating the size of each static reference. For more information, see the description in the short record format.
  - 1 bit indicating if the object was hashed and moved. If this bit is set, the record includes the hashcode.
- A “byte”, “short”, “int” or “long” containing the gap between the address of this class and the address of the preceding object. For more information, see the description in the short record format.
- An “int” containing the instance size.
- If all objects are hashed, a “short” containing the hashcode. Otherwise, an optional “int” containing the hashcode if the hashed and moved bit is set in the record flag byte.
- A “word” containing the address of the superclass.
- A UTF string containing the name of this class.
- An “int” containing the number of static references.
- The array of static references. For more information, see the description in the short record format.

Using the dump viewer

The SDK dump viewer presents system dump information in a readable format.

System dumps are produced in a platform-specific binary format. This format is typically a raw memory image of the process that was running at the time the dump was initiated. The dump viewer helps you navigate around the dump, obtaining information in a readable format. You can view Java information such as threads and objects on the heap, and native information such as native stacks, libraries, and raw memory locations.

The dump viewer is started with the **jdmpview** command. For detailed information about the dump viewer, see this section of the Java 6 diagnostics guide: http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/com.ibm.java.doc.diagnostics.60/diag/tools/dump_viewer_dtfjview/dump_viewer.html.

These topics contain additional information that applies to using the dump viewer with an IBM J9 2.6 virtual machine.

Support for compressed files

When you run the **jdmpview** tool on a compressed file, the tool detects and shows all system dump, Java dump, and heap dump files within the compressed file. Because of this behavior, more than one *context* might be displayed when you start **jdmpview**.

The context allows you to select which dump file you want to view. On z/OS, a system dump can contain multiple address spaces and multiple JVM instances. In this case, the context allows you to select the address space and JVM instance within the dump file.

If you do not use the **-core** or **-xml** options with the **-zip** option, **jdmpview** shows multiple contexts, one for each source file that it identified in the compressed file.

By default, when you specify a file by using the **-zip** option, the contents are extracted to a temporary directory before processing. Use the **-notemp** option to prevent this extraction step, and run all subsequent commands in memory. Because the commands are running in memory, you might have to increase the maximum heap size by using the **-Xmx** option, especially if you are analyzing a large heap.

Example 1

This example shows the output for a `.zip` file that contains a system dump from a Windows system. The example command to produce this output is `jdmpview -zip wintest.zip`:

```
Available contexts (* = currently selected context) :

Source : file:/C:/test/test.zip#core.20120402.151255.3728.0001.dmp
             *0 : PID: 3728 : JRE 1.6.0 Windows 7 amd64-64 build 20120119_100021 (pwa6460_26sr2-20120123_01(SR2))
```

Example 2

This example shows the output for a compressed file that contains a system dump from a z/OS system. The system dump contains multiple address spaces and two JVM instances:

```
Available contexts (* = currently selected context) :

Source : file:///D:/examples/MV2C.IVANPEG.D110706.T131828.S00053
         0 : ASID: 0x1 : No JRE : No JRE
         1 : ASID: 0x3 : No JRE : No JRE
         2 : ASID: 0x4 : No JRE : No JRE
         3 : ASID: 0x6 : No JRE : No JRE
         4 : ASID: 0x7 : No JRE : No JRE
```

```
        *5 : ASID: 0x73 EDB: 0x8004053a0 : JRE 1.6.0 z/OS s390x-64 build 20110217_75924
(pmz6460_26-20110228_01)
         6 : ASID: 0x73 EDB: 0x83d2053a0 : JRE 1.6.0 z/OS s390x-64 build 20110217_75924
(pmz6460_26-20110228_01)
         7 : ASID: 0x73 EDB: 0x4a7bd9e8 : No JRE
         8 : ASID: 0xffff : No JRE : No JRE
```

Example 3

This example shows the output for a compressed file that contains several system dump, Javadump, and Heapdump files:

```
Available contexts (* = currently selected context) :

Source : file:/D:/Samples/multi-image.zip#core1.dmp
           *0 : PID: 10463 : JRE 1.6.0 Linux amd64-64 build 20120228_104045 pxa6460_26sr1-20120302_01(SR2))

Source : file:/D:/Samples/multi-image.zip#core2.dmp
              1 : PID: 12268 : JRE 1.6.0 Linux amd64-64 build 20120228_104045 pxa6460_26sr1-20120302_01(SR2))

Source : file:/D:/Samples/multi-image.zip#javacore.20120228.100128.10441.0002.txt
              2 : JRE 1.6.0 Linux amd64-64 build 20120228_94967 (pxa6460_26sr1-20120302_01(SR2))

Source : file:/D:/Samples/multi-image.zip#javacore.20120228.090916.14653.0002.txt
              3 : JRE 1.6.0 Linux amd64-64 build 20120228_94967 (pxa6460_26sr1-20120302_01(SR2))

Source : file:/D:/Samples/multi-image.zip#heapdump.20111115.093819.4336.0001.phd
              4 : JRE 1.6.0 Windows 7 amd64-64 build 20111105_94283 (pxa6460_26sr1-20120302_01(SR2))
```

Working with Java dump and heap dump files

When working with Java dump and heap dump files, some **jdmpview** commands do not produce any output. This result is because Java dump files contain only a summary of JVM and native information (excluding the contents of the Java heap), and heap dump files contain only summary information about the Java heap. See Example 3 listed previously; context 4 is derived from a heap dump file:

```
Source : file:/D:/Samples/multi-image.zip#heapdump.20111115.093819.4336.0001.phd
              4 : JRE 1.6.0 Windows 7 amd64-64 build 20111105_94283 (pxa6460_26sr1-20120302_01(SR2))
```

If you select this context, and run the **info system** command, some data is shown as unavailable:

```
CTX:0> context 4
CTX:4> info system
Machine OS:     Windows 7
Machine name:   data unavailable
Machine IP address(es):
                data unavailable
System memory:  data unavailable
```

However, because this context is for a heap dump file, the **info class** command can provide a summary of the Java heap:

```
CTX:4> info class
instances total size class name
0 0 sun/io/Converters
1 16 com/ibm/tools/attach/javaSE/FileLock$syncObject
2 32 com/ibm/tools/attach/javaSE/AttachHandler$syncObject
1 40 sun/nio/cs/UTF_16LE
....
Total number of objects: 6178
Total size of objects: 2505382
```

Processing system dumps

Some system dumps must be processed before you can examine them with the dump viewer.

To analyze system dumps from Linux and AIX platforms, copies of executable files and libraries are required along with the system dump. You must run the **jextract** utility or the Diagnostic Collector provided in the SDK to collect these files on the machine that produced the system dump. The parameters for the **jextract** command are:

```
jextract <dump_file_name> [<zip_file>]
```

This command generates a compressed (.zip) file containing the system dump and the required executable file and libraries.

For system dumps generated from a IBM J9 2.6 virtual machine in IBM SDK, Java Technology Edition, Version 6 on Windows and z/OS platforms, you no longer need to run the **jextract** tool.

For system dumps generated from an IBM J9 2.4 virtual machine in Java v6, or from an IBM J9 2.3 virtual machine in Java v5.0, you must continue to run the **jextract** utility for all platforms.

Dump viewer: `jdmpview`

The dump viewer is a utility supplied in the SDK that allows you to examine the contents of system dumps. For system dumps generated from a IBM J9 2.6 virtual machine in IBM SDK, Java Technology Edition, Version 6, you no longer need to specify a metadata file when using the dump viewer. The parameters for the **jdmpview** command are:

```
jdmpview -core <system dump file> [-xml <xml file>]
```

-core <*system dump file*>
Specify a system dump filename.

-xml <*xml file*>
Specify a dump metadata file. Not required for system dumps generated from an IBM J9 2.6 virtual machine in IBM SDK, Java Technology Edition, Version 6.

Using the dump viewer in batch mode

For long running or routine jobs, **jdmpview** can be used in batch mode.

You can run a single command without specifying a command file by appending the command to the end of the **jdmpview** command line. For example:

```
jdmpview -core mycore.dmp info class
```

When specifying **jdmpview** commands that accept a wildcard parameter, you must replace the wildcard symbol with `ALL` to prevent the shell interpreting the wildcard symbol. For example, in interactive mode, the command `info thread *` must be specified as:

```
jdmpview -core mycore.dmp info thread ALL
```

Batch mode is controlled with the following command-line options:

-cmdfile <*path to command file*>
A file containing a series of **jdmpview** commands. These commands are read and run sequentially.

-charset <*character set name*>
The character set for the commands specified in **-cmdfile**.

The character set name must be a supported *charset* as defined in java.nio.charset.Charset. For example, `US-ASCII`.

-outfile *<path to output file>*
The file to record any output generated by commands.

-overwrite
If the file specified in **-outfile** exists, this option overwrites the file.

Consider a command file, `commands.txt` with the following entries:

```
info system
info proc
```

The **jdmpview** command can be run in the following way:

```
jdmpview -outfile out.txt [-overwrite] -cmdfile commands.txt -core <path to core file>
```

An error message is shown if the output file exists and you do not specify the **-overwrite** option.

The following output is shown in the console and in the output file, `out.txt`:

```
DTFJView version 2.1.1, using DTFJ API version 1.7
Loading image from DTFJ...

Available contexts (* = currently selected context) :

*0 : PID: 14279 : JRE 1.6.0 Linux x86-32 build 20111001_91728 (pxi3260_26sr1-20111004_01(SR1)

> info system

Machine OS: Linux
Machine name: mysystem
Machine IP address(es):
  127.0.1.1
System memory: 4146188288

Java version :

Java(TM) SE Runtime Environment(build JRE 1.6.0 Linux x86-32 build 20111001_91728 (pxi3260_26sr1-
20111004_01(SR1)))
IBM J9 VM(JRE 1.6.0 IBM J9 2.6 Linux x86-32 20111001_91728 (JIT enabled, AOT enabled)
J9VM - R26_JVM_26_20110930_1540_B91699
JIT  - r11_20110916_20778
GC   - R26_JVM_26_20110923_1426_B91192)

> info proc

  Native thread IDs:
   5854 5871

  Command line arguments
   java -Xdump:system:events=vmstop -version

  Environment variables:
   DISPLAY=:0.0
   PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
   COLORTERM=gnome-terminal
   DESKTOP_SESSION=gnome-classic
   SHELL=/bin/bash
          OLDPWD=/home/bczapp
   LANGUAGE=en_GB:en
   WINDOWID=77594661
   LANG=en_GB.UTF-8
   GDM_KEYBOARD_LAYOUT=gb
   IBM_JAVA_COMMAND_LINE=java -Xdump:system:events=vmstop -version
```

```
|     GDM_LANG=en_GB
|     TERM=xterm
|     _=/usr/bin/java
|            ...
```

Commands available in `jdmpview`

jdmpview 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
- Thread properties: list of native thread properties and their values. For example: thread priority.
- Associated Java thread, if applicable:
  - Name of Java thread
  - Address of associated java.lang.Thread object
  - State (shown in JVMTI and java.lang.Thread.State formats)
  - The monitor the thread is waiting for
  - 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*>] [-sort:<*name*>|<*count*>|<*size*>**]**
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 class name
- the total number of instances of all classes.
- the total size of all objects.

The `-sort` option allows the list of classes to be sorted by name (default), by number of instances of each class, or by the total size of instances of each class.

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 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 [<*address*>**] [-verbose] [-sort:**<*size*>|<*address*>**]**
Displays a summary list of memory sections in the process address space, with start and end address, size, and properties. If an address parameter is specified, the results show details of only the memory section containing the address. If `-verbose` is specified, full details of the properties of each memory section are displayed. The `-sort` option allows the list of memory sections to be sorted by size or by start address (default).

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

heapdump [<*heaps*>]
Generates a Heapdump to a file. You can select which Java heaps to dump by listing the heap names, separated by spaces. To see which heaps are available, use the **info heap** command. By default, all Java heaps are dumped.

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 heapdump <*options*>
Configures Heapdump generation settings.

The options are:

phd
Set the Heapdump format to Portable Heapdump, which is the default.

txt
Set the Heapdump format to classic.

file <*file*>
Set the destination of the Heapdump.

multiplefiles [on|off]
If **multiplefiles** is set to `on`, each Java heap in the system dump is written to a separate file. If **multiplefiles** is set to `off`, all Java heaps are written to the same file. The default is `off`.

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 heapdump <*options*>
Displays the current Heapdump generation settings.

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:

```
-------------------------------------------------------------------
> whatis 0x8e76a8

heap #1 - name: Default@19fce8
0x8e76a8 is within heap segment: 8b0000 -- cb0000
0x8e76a8 is start of an object of type java/lang/Thread
-------------------------------------------------------------------
```

x/ (examine)

Passes the number of items to display and the unit size, as listed in the following table, to the sub-command. For example, `x/12bd`. This command is similar to the use of the **x/** command in **gdb**, including the use of defaults.

Table 8. Unit sizes

Abbreviation	Unit	Size
b	Byte	8-bit
h	Half word	16-bit
w	Word	32-bit
g	Giant word	64-bit

x/J [<*class_name*>|<*0xaddr*>]

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.

Working with dumps containing multiple JVMs

On z/OS, system dumps can contain multiple address spaces. Multiple JVMs can also share a single address space. The **jdmpview** command lets you work with these dumps by separating the dump into contexts.

Start **jdmpview** to see a list of available contexts:

```
CTX:5> context
Available contexts (* = currently selected context) :

 0 : ASID: 0x1 : No JRE : No JRE
 1 : ASID: 0x3 : No JRE : No JRE
 2 : ASID: 0x4 : No JRE : No JRE
 3 : ASID: 0x6 : No JRE : No JRE
 4 : ASID: 0x7 : No JRE : No JRE
*5 : ASID: 0x73 EDB: 0x83d2053a0 : JRE 1.6.0 z/OS s390x-64 build 20110217_75924 (pmz6460_26-20110228_01)
 6 : ASID: 0x73 EDB: 0x8004053a0 : JRE 1.6.0 z/OS s390x-64 build 20110217_75924 (pmz6460_26-20110228_01)
 7 : ASID: 0x73 EDB: 0x4a7bd9e8 : No JRE
 8 : ASID: 0xffff : No JRE : No JRE

CTX:5>
```

Each address space (ASID) is listed as a separate context. Each JVM occupying a single address space is also listed as a separate context. In the example, contexts `5` and `6` in the address space `0x73` are separate JVMs. The prompt `CTX:5>` indicates the context that is currently in use. If you run a command at the prompt, the action is applied to the JVM occupying the current context.

Run the **context** command to see all available contexts, including JVM build information for JVMs, where appropriate:

```
CTX:5> context
Available contexts (* = currently selected context) :

 0 : ASID: 0x1 : No JRE : No JRE
 1 : ASID: 0x3 : No JRE : No JRE
 2 : ASID: 0x4 : No JRE : No JRE
 3 : ASID: 0x6 : No JRE : No JRE
 4 : ASID: 0x7 : No JRE : No JRE
*5 : ASID: 0x73 EDB: 0x83d2053a0 :
  Java(TM) SE Runtime Environment(build JRE 1.6.0 z/OS s390x-64 build 20110217_75924 (pmz6460_26-
20110228_01))
  IBM J9 VM(JRE 1.6.0 IBM J9 2.6 z/OS s390x-64 20110217_75924 (JIT enabled, AOT enabled)
  J9VM - R26_Java626_GA_20110217_1713_B75924
  JIT  - r11_20110215_18645
  GC   - R26_Java626_GA_20110217_1713_B75924)
```

```
  6 : ASID: 0x73 EDB: 0x8004053a0 :
   Java(TM) SE Runtime Environment(build JRE 1.6.0 z/OS s390x-64 build 20110217_75924 (pmz6460_26-
20110228_01))
   IBM J9 VM(JRE 1.6.0 IBM J9 2.6 z/OS s390x-64 20110217_75924 (JIT enabled, AOT enabled)
   J9VM - R26_Java626_GA_20110217_1713_B75924
   JIT  - r11_20110215_18645
   GC   - R26_Java626_GA_20110217_1713_B75924)
  7 : ASID: 0x73 EDB: 0x4a7bd9e8 : No JRE
  8 : ASID: 0xffff : No JRE : No JRE
```

You can switch between contexts by typing **context** <*n*>, where <*n*> is the context you want to switch to. For example:

```
CTX:5> context 6
CTX:6>
```

Following this switch, the output from the context command is:

```
CTX:6> context
Available contexts (* = currently selected context) :

  0 : ASID: 0x1 : No JRE : No JRE
  1 : ASID: 0x3 : No JRE : No JRE
  2 : ASID: 0x4 : No JRE : No JRE
  3 : ASID: 0x6 : No JRE : No JRE
  4 : ASID: 0x7 : No JRE : No JRE
  5 : ASID: 0x73 EDB: 0x83d2053a0 :
   Java(TM) SE Runtime Environment(build JRE 1.6.0 z/OS s390x-64 build 20110217_75924 (pmz6460_26-
20110228_01))
   IBM J9 VM(JRE 1.6.0 IBM J9 2.6 z/OS s390x-64 20110217_75924 (JIT enabled, AOT enabled)
   J9VM - R26_Java626_GA_20110217_1713_B75924
   JIT  - r11_20110215_18645
   GC   - R26_Java626_GA_20110217_1713_B75924)
*6 : ASID: 0x73 EDB: 0x8004053a0 :
   Java(TM) SE Runtime Environment(build JRE 1.6.0 z/OS s390x-64 build 20110217_75924 (pmz6460_26-
20110228_01))
   IBM J9 VM(JRE 1.6.0 IBM J9 2.6 z/OS s390x-64 20110217_75924 (JIT enabled, AOT enabled)
   J9VM - R26_Java626_GA_20110217_1713_B75924
   JIT  - r11_20110215_18645
   GC   - R26_Java626_GA_20110217_1713_B75924)
  7 : ASID: 0x73 EDB: 0x4a7bd9e8 : No JRE
  8 : ASID: 0xffff : No JRE : No JRE
```

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 trace points at run time by using levels, components, group names, or individual trace point identifiers.

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

Default tracing

There are some changes to default assertion tracing for the IBM J9 2.6 virtual machine.

Default assertion tracing

The JVM includes assertions, implemented as special trace points. By default, internal assertions are detected and diagnostic logs are produced to help assess the error.

Assertion failures often indicate a serious problem, and the JVM usually stops immediately. Send a service request to IBM, including the standard error output and any diagnostic files that are produced.

When an assertion trace point is reached, 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
```

Assertion failures might occur early during JVM startup, before trace is enabled. In this case, the assert message has a different format, and is not prefixed by a timestamp or thread ID. For example:

```
** ASSERTION FAILED ** j9vmutil.15 at thrinfo.c:371 Assert_VMUtil_true((
publicFlags & 0x200))
```

Assertion failures that occur early during startup cannot be disabled. These failures do not produce diagnostic dumps, and do not cause the JVM to stop.

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.TraceFormat <input_file>  [<output_file>] [options]
```

where *<input_file>* is the name of the binary trace file to be formatted, and *<output_file>* is the name of the output file.

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:

-datfile=<*file1.dat*>[,<*file2.dat*>]
A comma-separated list of trace formatting data files. By default, the files used are:`$JAVA_HOME/lib/J9TraceFormat.dat` and `$JAVA_HOME/lib/TraceFormat.dat`

-format_time=yes|no
Specifies whether to format the time stamps into human readable form. The default is `yes`.

-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.

-summary
Prints summary information to the screen without generating an output file.

-threads=<*thread id*>[,<*thread id*>]...
Filters the output for the given thread IDs only. *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.

-timezone=+|-HH:MM
Specifies the offset from UTC, as positive or negative hours and minutes, to apply when formatting timestamps.

-verbose
Output detailed warning and error messages, and performance statistics.

Shared classes diagnostic data

Understanding how to diagnose problems that might occur helps you to use shared classes mode.

Deploying shared classes

There are some changes to class data sharing that require consideration.

When you enable class data sharing, there are a number of deployment considerations. These considerations are detailed in the IBM SDK, Java Technology Edition, Version 6 Diagnostics Guide, see http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/com.ibm.java.doc.diagnostics.60/diag/tools/shcpd_deploying.html. For this release, there are some changes to class data sharing that can have additional implications on cache performance.

Cache naming:

Cache naming enhancements include the ability to use the **-Xshareclasses:cacheDirPerm=**<*permission*> suboption to specify permissions for the cache directory.

If you use the **-Xshareclasses:cachedir=**<*dir*> suboption to specify a cache directory that does not already exist, you can also use the **-Xshareclasses:cacheDirPerm=**<*permission*> suboption to specify permissions for the directory when it is created. This suboption is available only on AIX, UNIX and z/OS operating systems. You can use this suboption to restrict access to the cache directory, however this suboption can conflict with the **groupAccess** suboption, which is used to set permissions on a cache. The **cachedir** suboption also affects the permissions of persistent caches. For more information about **-Xshareclasses** suboptions, see "-Xshareclasses" on page 155.

Cache performance:

Performance improvements include the ability to cache JIT data, and the ability to reserve a portion of the cache for specific activities.

Caching JIT data

The JVM can automatically store a small amount of JIT data in the cache when it is populated with classes. The JIT data enables any subsequent JVMs attaching to the cache to either start faster, run faster, or both.

You can use the **-Xshareclasses:nojitdata**, **-Xscminjitdata<size>**, and **-Xscmaxjitdata<size>** options to control the use of JIT data in the cache.

JIT data is associated with 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 JIT data can be generated for those classes. If a particular class becomes stale, the JIT data associated with that class also becomes stale. If a class is redeemed, the JIT data associated with that class is also redeemed. JIT data is not shared between multiple versions of the same class.

The total amount of JIT data can be limited using **-Xscmaxjitdata**<*size*>, and cache space can be reserved for JIT data using **-Xscminjitdata**<*size*>.

In general, the default settings provide significant performance benefits and use only a small amount of cache space. However, if you want to prevent the JVM storing any JIT data, you can specify **-Xshareclasses:nojitdata**.

Class Debug Area

A portion of the shared classes cache is reserved for storing the class attribute information LineNumberTable and LocalVariableTable, which are used for printing stack traces and for Java debugging. By storing these attributes in a separate region, the operating system can decide whether to keep the region in memory or on disk, depending on whether the data is being used.

You can control the size of the Class Debug Area using the **-Xscdmx** command-line option. Use any of the following variations to specify a Class Debug Area with a size of 1 MB:

- **-Xscdmx**1048576
- **-Xscdmx**1024k
- **-Xscdmx**1m

The number of bytes passed to **–Xscdmx** must always be less than the total cache size. This value is always rounded down to the nearest multiple of the system page size.

The amount of LineNumberTable and LocalVariableTable attribute information stored for different applications varies. When the Class Debug Area is full, use **-Xscdmx** to increase the size. When the Class Debug Area is not full, create a smaller region, which increases the available space for other artifacts elsewhere in the cache.

The size of the Class Debug Area affects available space for other artifacts, like AOT code, in the shared classes cache. Performance might be adversely affected if the cache is not sized appropriately. You can improve performance by using the **-Xscdmx** option to resize the Class Debug Area, or by using the **-Xscmx** option to create a larger cache.

If you start the JVM with **-Xnolinenumbers** when creating a new shared classes cache, the Class Debug Area is not created. The option **-Xnolinenumbers** advises

the JVM not to load any class debug information, so there is no need for this region. If **-Xscdmx** is also used on the command line to specify a non zero debug area size, then a debug area is created despite the use of **-Xnolinenumbers**.

Raw Class Data Area

When a cache is created with **-Xshareclasses:enableBCI**, a portion of the shared classes cache is reserved for storing the original class data bytes. Storing this data in a separate region allows the operating system to decide whether to keep the region in memory or on disk, depending on whether the data is being used. Because the amount of raw class data stored in this area can vary for an application, the size of the Raw Class Data Area can be modified using the **rcdSize** suboption. For example, these variations specify a Raw Class Data Area with a size of 1 MB:

```
-Xshareclasses:enableBCI,rcdSize=1048576
-Xshareclasses:enableBCI,rcdSize=1024k
-Xshareclasses:enableBCI,rcdSize=1m
```

The number of bytes passed to **rcdSize** must always be less than the total cache size. This value is always rounded down to the nearest multiple of the system page size. As with the Class Debug Area, the size of this area affects available space for other artifacts, such as AOT code, in the shared classes cache. Performance might be adversely affected if the cache is not sized appropriately. When the cache is created without **enableBCI**, the default size of the Raw Class Data Area is 0 bytes. However, when the **enableBCI** is used, a portion of the cache is automatically reserved.

Dealing with runtime bytecode modification

Modifying bytecode at run time is a 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 several considerations to avoid potential problems. These considerations are detailed in the IBM SDK, Java Technology Edition, Version 6 Diagnostics Guide, see http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/com.ibm.java.doc.diagnostics.60/diag/tools/shcpd_runtime_bytecode_mod.html. For this release, there are new features that require further consideration.

Using the JVMTI ClassFileLoadHook with cached classes:

The **-Xshareclasses:enableBCI** suboption improves startup performance without using a modification context, when using JVMTI class modification. This suboption allows classes loaded from the shared cache to be modified using a JVMTI ClassFileLoadHook, or a java.lang.instrument agent, and prevents modified classes being stored in the shared classes cache.

Modification contexts allow classes modified at run time by JVMTI agents to be stored, logically separated, in the cache. This separation prevents conflicts with versions of the same class that are being used by other JVMs connected to the cache. However, there are a number of issues:

- Loading classes from the cache does not generate a callback to the JVMTI ClassFileLoadHook event, which prevents a JVMTI agent making any subsequent modifications. The ClassFileLoadHook event expects original class data to be passed back. This data is typically not available in the shared cache unless the cache was created with a JVMTI agent that is retransformation

capable. This constraint might be undesirable for JVMTI or java.lang.instrument agents that want the ClassFileLoadHook event to be triggered every time, whether the class is loaded from the cache, or from the disk.

- If the JVMTI agent applies different runtime modifications every time the application is run, there will be multiple versions of the same class in the cache that cannot be reused or shared across JVMs.

To address these issues, use the suboption **-Xshareclasses:enableBCI**. When using this suboption, any class modified by a JVMTI or java.lang.instrument agent is not stored in the cache. Classes which are not modified are stored as before. The **-Xshareclasses:enableBCI** suboption causes the JVM to store original class byte data in the cache, which allows the ClassFileLoadHook event to be triggered for all classes loaded from the cache. When using this suboption, the cache size might need to be increased with **-Xscmx<size>**.

Using this option can improve the startup performance when JVMTI agents, java.lang.instrument agents, or both, are being used to modify classes. If you do not use this option, the JVM is forced to load classes from disk and find the equivalent class in the shared cache by doing a comparison. Because loading from disk and class comparison is done for every class loaded, the startup performance can be affected. Using **-Xshareclasses:enableBCI** loads unmodified classes directly from the shared cache, improving startup performance, while still allowing these classes to be modified by the JVMTI agents, java.lang.instrument agents, or both.

Using **-Xshareclasses:enableBCI** with a modification context is still valid. However, **-Xshareclasses:enableBCI** prevents modified classes from being stored in the cache. Although unmodified classes are stored in the cache and logically separated by the specified modification context, using a modification context with **-Xshareclasses:enableBCI** does not provide any benefits and should be avoided.

When a new shared cache is created with **-Xshareclasses:enableBCI**, a portion of the shared cache is reserved for storing the original class data in the shared classes cache. Storing this data in a separate region allows the operating system to decide whether to keep the region in memory or on disk, depending on whether the data is being used. When this area is full, the original class data is stored with the rest of the shared class data. For more information about this area, known as the Raw Class Data Area, see “Cache performance” on page 102.

Using the Java Helper API

Classes are shared by the bootstrap class loader internally in the JVM. 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 so that Java class loaders can use the shared classes features in the JVM.

The Helper API classes are contained in the com.ibm.oti.shared package. Further information about the Java Helper API is provided in the Java 6 Diagnostics guide. See http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/com.ibm.java.doc.diagnostics.60/diag/tools/shcpd_helper.html.

Utility APIs:

Use these APIs to obtain information about shared caches.

com.ibm.oti.shared.SharedClassUtilities
You can use these APIs to get information about shared class caches in a directory, and to remove specified shared class caches. The type of information available for each cache includes:

- The cache name.
- The cache size.
- The amount of free space in the cache.
- An indication of compatibility with the current JVM.
- Information about the type of cache; persistent or non-persistent.
- The last detach time.
- The Java version that created the cache.
- Whether the cache is for a 32-bit or 64-bit JVM.
- Whether the cache is corrupted.

com.ibm.oti.shared.SharedClassCacheInfo
This class is used by com.ibm.oti.shared.SharedClassUtilities to store information about a shared class cache and provides API methods to retrieve that information.

For information about the related IBM JVMTI extensions for shared class caches, see "Finding shared class caches" on page 129, and "Removing a shared class cache" on page 131.

Understanding shared classes diagnostic output

When running in shared classes mode, a number of diagnostic tools can help you. The verbose options are used at run time 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.

printStats utility:

The **printStats** utility prints summary information about the specified cache to the standard error output. Information about zip caches, the amount of JIT data stored, and the size of the class debug area, is additional to the information provided for IBM SDK, Java Technology Edition, Version 6. You can optionally specify one or more types of cache content, such as AOT data or tokens, to see more detailed information about that type of content. To see detailed information about all the types of content in the cache, use the **printAllStats** utility instead.

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 without a parameter, to generate summary data only:

```
Cache created with:
-Xnolinenumbers = false
BCI Enabled = true

Cache contains only classes with line numbers

base address = 0x00002AAACE282000
end address = 0x00002AAACF266000
allocation pointer = 0x00002AAACE3A61B0
```

```
cache size = 16776608
free bytes = 6060232
ROMClass bytes = 1196464
AOT bytes = 0
Reserved space for AOT bytes = -1
Maximum space for AOT bytes = -1
JIT data bytes = 0
Reserved space for JIT data bytes = -1
Maximum space for JIT data bytes = -1
Zip cache bytes = 1054352
Data bytes = 114080
Metadata bytes = 24312
Metadata % used = 1%
Class debug area size = 1331200
Class debug area used bytes = 150848
Class debug area % used = 11%
Raw class data area size = 6995968
Raw class data used bytes = 1655520
Raw class data area % used = 23%

# ROMClasses = 488
# AOT Methods = 0
# Classpaths = 1
# URLs = 0
# Tokens = 0
# Zip caches = 22
# Stale classes = 0
% Stale classes = 0%

Cache is 28% full
```

In the example output, `-Xnolinenumbers = false` means the cache was created without the **-Xnolinenumbers** option being specified.

`BCI Enabled = true` indicates that the cache was created with the **-Xshareclasses:enableBCI** suboption.

One of the following messages is displayed to indicate the line number status of classes in the shared cache:

Cache contains only classes with line numbers
JVM line number processing was enabled (the **-Xnolinenumbers** option was not specified) for all the classes that were put in this shared cache. All classes in the cache contain line numbers if the original classes contained line number data.

Cache contains only classes without line numbers
JVM line number processing was disabled (the **-Xnolinenumbers** option was specified) for all the classes that were put in this shared cache, so none of the classes contain line numbers.

Cache contains classes with line numbers and classes without line numbers
JVM line number processing was enabled for some classes and disabled for others (the **-Xnolinenumbers** option was specified when some of the classes were added to the cache).

The following summary data is displayed:

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.

Reserved space for AOT bytes
The number of bytes reserved for AOT compiled code in the cache.

Maximum space for AOT bytes
The maximum number of bytes of AOT compiled code that can be stored in the cache.

JIT data bytes
Is the number of bytes of JIT-related data stored in the cache.

Reserved space for JIT data bytes
The number of bytes reserved for JIT-related data in the cache.

Maximum space for JIT data bytes
The maximum number of bytes of JIT-related data that can be stored in the cache.

Zip cache bytes
Is the number of zip entry cache bytes stored 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. The value shown does consider the `Class debug area size`.

Class debug area size
Is the size in bytes of the Class Debug Area. This area is reserved to store LineNumberTable and LocalVariableTable class attribute information.

Class debug area bytes used
Is the size in bytes of the Class Debug Area that contains data.

Class debug area % used
Is the percentage of the Class Debug Area that contains data.

Raw class data area size
The size in bytes of the Raw Class Data Area. This area is reserved when the cache is created with **-Xshareclasses:enableBCI**, or **-Xshareclasses:rcdSize=nnn**. The original class file bytes for a ROMClass are stored here when `enableBCI` is used to create the cache.

Raw class data used bytes
The size in bytes of the Raw Class Data Area that contains data.

Raw class data area % used
The percentage of the Raw Class Data Area that contains data.

 # 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 105.

 # 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 class loaders, including the bootstrap and application class loaders, 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. For more information about partitions, see ../../../../com.ibm.java.doc.diagnostics.60/diag/tools/shcpd_rbm_partitions.html. For more information about modification contexts, see ../../../../com.ibm.java.doc.diagnostics.60/diag/tools/shcpd_rbm_contexts.html.

 # Zip caches
The number of .zip files that have entry caches stored in the shared cache.

 # Stale classes
Are classes that have been marked as "potentially stale" by the cache code, because of updates to Java classes. See ../../../../com.ibm.java.doc.diagnostics.60/diag/tools/shcpd_dynamic.html.

% Stale classes
Is an indication of the proportion of classes in the cache that have become stale.

Cache is XXX% full
Shows the percentage of the cache that is currently used. The value displayed does not consider the `Class debug area size`. The calculation for this value is:

```
% Full = (('Cache Size' - 'Debug Area Size' - 'Free Bytes') * 100) / ('Cache Size' - 'Debug Area Size')
```

Generating more detailed information

You can use a parameter to specify one or more types of cache content. The printStats utility then provides more detailed information about that type of content, in addition to the summary data described previously. The detailed output is similar to the output from the printAllStats utility. For more information about the different types of cache content and the printAllStats utility, see "printAllStats utility."

If you want to specify more than one type of cache content, use the plus symbol (+) to separate the values:

```
printStats[=type_1[+type_2][...]]
```

For example, use `printStats=classpath` to see a list of class paths that are stored in the shared cache, or `printStats=romclass+url` to see information about ROMClasses and URLs.

The following data types are valid. The values are not case sensitive:

Help Prints a list of valid data types.

All Prints information about all the following data types in the shared cache. This output is equivalent to the output produced by the printAllStats utility.

Classpath
Lists the class paths that are stored in the shared cache.

URL Lists the URLs that are stored in the shared cache.

Token Lists the tokens that are stored in the shared cache.

ROMClass
Prints information about the ROMClasses that are stored in the shared cache. This parameter does not print information about ROMMethods in ROMClasses.

ROMMethod
Prints ROMClasses and the ROMMethods in them.

AOT Prints information about AOT compiled code in the shared cache.

JITprofile
Prints information about JIT data in the shared cache.

JIThint
Prints information about JIT data in the shared cache.

ZipCache
Prints information about zip entry caches that are stored in the shared cache.

printAllStats utility:

The printAllStats utility is a suboption of **-Xshareclasses**, optionally taking a cache name using **name=**<*name*>. Information about zip caches and JIT data is additional to the information provided for IBM SDK, Java Technology Edition, Version 6.

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. For more information about this option, see "-Xzero" on page 151.

JIT data

```
1: 0xD6290368 JITPROFILE: getKeyHash Signature: ()I Address: 0xD55118C0
for ROMClass java/util/Hashtable$Entry at 0xD5511640.
2: 0xD6283848 JITHINT: loadClass Signature: (Ljava/lang/String;)Ljava/lang/Class; Address: 0xD5558F98
for ROMClass com/ibm/oti/vm/BootstrapClassLoader at 0xD5558AE0.
```

The JIT stores data in the shared classes cache in the form of `JITPROFILE` and `JITHINT` entries to improve runtime performance. These outputs expose the content of the shared cache and can be useful for diagnostic purposes.

For more information, see the Diagnostic Guide topic about the printAllStats utility provided with IBM SDK, Java Technology Edition, Version 6.

Garbage Collector diagnostic data

This section describes changes to garbage collection diagnostic data.

Verbose garbage collection logging

Verbose garbage collection logging has been redesigned, improving problem diagnosis.

With earlier releases of the IBM SDK, JavaTechnology Edition, verbose logging generates a summary of garbage collection at the end of a full garbage collection cycle. In contrast, the new verbose logging function is event-based, generating data for each garbage collection operation, as it happens.

A garbage collection cycle is made up of one or more garbage collection operations, spread across one or more garbage collection increments. A garbage collection cycle can be caused by a number of events, including:

- Calls to System.gc().
- Allocation failures.
- Completing concurrent collections.
- Decisions based on the cost of making resource allocations.

The verbose garbage collection output for each event contains an incrementing ID tag. The ID increments for each event, regardless of event type, so you can use this tag to search within the output for specific events.

By default, **-verbose:gc** output is written to stderr. You can redirect the information to a file by using the **-Xverbosegclog** command-line option, see Garbage Collection command line options.

The contents of the **-verbose:gc** output might change from release to release, when improvements are made to the technology or when new data becomes available.

New IBM JVMTI extensions are available for subscribing to, and unsubscribing from, verbose garbage collection logging. For information about using these extensions, see "IBM JVMTI extensions - API reference" on page 126.

The following sections show sample results for different garbage collection events.

Garbage collection initialization:

When garbage collection is initialized, verbose logging generates output showing the garbage collection options in force. These items can be modified with options such as **-Xgcthreads**.

The first tag shown in the output is the `<initialized>` tag, which is followed by values that include an `id` and `timestamp`. The information shown in the `<initialized>` section includes the garbage collection policy, the policy options, and any JVM command-line options that are in effect at the time.

```
<initialized id="1" timestamp="2010-11-23T00:41:32.328">
  <attribute name="gcPolicy" value="-Xgcpolicy:gencon" />
  <attribute name="maxHeapSize" value="0x5fcf0000" />
  <attribute name="initialHeapSize" value="0x400000" />
  <attribute name="compressedRefs" value="false" />
  <attribute name="pageSize" value="0x1000" />
  <attribute name="requestedPageSize" value="0x1000" />
  <attribute name="gcthreads" value="2" />
  <system>
    <attribute name="physicalMemory" value="3214884864" />
    <attribute name="numCPUs" value="2" />
    <attribute name="architecture" value="x86" />
    <attribute name="os" value="Windows XP" />
    <attribute name="osVersion" value="5.1" />
  </system>
  <vmargs>
    <vmarg name="-Xoptionsfile=C:\jvmwi3270\jre\bin\default\options.default" />
    <vmarg name="-Xlockword:mode=default,noLockword=java/lang/String,noLockword=
     java/util/MapEntry,noLockword=java/util/HashMap$Entry,noLockword..." />
    <vmarg name="-XXgc:numaCommonThreadClass=java/lang/UNIXProcess$*" />
    <vmarg name="-Xjcl:jclscar_26" />
    <vmarg name="-Dcom.ibm.oti.vm.bootstrap.library.path=C:\jvmwi3270\jre\bin\
     default;C:\jvmwi3270\jre\bin" />
    <vmarg name="-Dsun.boot.library.path=C:\jvmwi3270\jre\bin\default;C:\
     jvmwi3270\jre\bin" />
    <vmarg name="-Djava.library.path=C:\jvmwi3270\jre\bin\default;C:\
     jvmwi3270\jre\bin;.;c:\pwi3260\jre\bin;c:\pwi3260\bin;C:\WINDOWS\syst..." />
    <vmarg name="-Djava.home=C:\jvmwi3270\jre" />
    <vmarg name="-Djava.ext.dirs=C:\jvmwi3270\jre\lib\ext" />
    <vmarg name="-Duser.dir=C:\jvmwi3270\jre\bin" />
    <vmarg name="_j2se_j9=1119744" value="7FA9CEF8" />
    <vmarg name="-Dconsole.encoding=Cp437" />
    <vmarg name="-Djava.class.path=." />
    <vmarg name="-verbose:gc" />
    <vmarg name="-Dsun.java.command=Foo" />
    <vmarg name="-Dsun.java.launcher=SUN_STANDARD" />
    <vmarg name="_port_library" value="7FA9C5D0" />
    <vmarg name="_bfu_java" value="7FA9D9BC" />
    <vmarg name="_org.apache.harmony.vmi.portlib" value="000AB078" />
  </vmargs>
</initialized>
```

Stop-the-world operations:

When an application is stopped so that the garbage collector has exclusive access to the Java virtual machine verbose logging records the event.

```
<exclusive-start id="3663" timestamp="2015-12-07T14:25:14.704" intervalms="188.956">
<response-info timems="0.131" idlems="0.048" threads="3" lastid="000000000258CE00" lastname="Pooled Thread #2"/>
</exclusive-start>
......
<exclusive-end id="3674" timestamp="2015-12-07T14:25:14.732" durationms="27.513" />
```

The items in this section of the log are explained as follows:

<exclusive-start> and <exclusive-end>
These tags represent a stop-the-world operation. The tags have the following attributes:

timestamp
The local timestamp at the start or end of the stop-the-world operation.

<response-info>
This tag provides details about the process of acquiring exclusive access to the virtual machine. This tag has the following attributes:

timems The time, in milliseconds, that was taken to acquire exclusive access to the virtual machine. To obtain exclusive access, the garbage collection thread requests all other threads to stop processing, then waits for those threads to respond to the request. If this time is excessive, you can use the **-Xdump:system:events** command-line parameter to create a system dump. The dump file might help you to identify threads that are slow to respond to the exclusive access request. For example, the following option creates a system dump when a thread takes longer than one second to respond to an internal virtual machine request:

```
-Xdump:system:events=slow,filter=1000ms
```

idlems 'Idle time' is the time between one of the threads responding and the final thread responding. During this time, the first thread is waiting, or 'idle'. The reported time for idlems is the mean idle time (in milliseconds) of all threads.

threads
The number of threads that were requested to release VM access. All threads must respond.

lastid The last thread to respond.

lastname
The name of the thread that is identified by the `lastid` attribute.

durationms
The total time for which exclusive access was held by the garbage collection thread.

Garbage collection cycle:

Verbose garbage collection output shows each garbage collection cycle enclosed within `<cycle-start>` and `<cycle-end>` tags. Each garbage collection cycle includes at least one garbage collection increment.

The <cycle-end> tag contains a context-id attribute that matches the id of the corresponding <cycle-start> tag.

```
<cycle-start id="4" type="scavenge" contextid="0" timestamp="2010-11-23T00:41:32.515" intervalms="225.424" />
<cycle-end id="10" type="scavenge" contextid="4" timestamp="2015-12-07T14:21:11.421" />
```

In the example, the <cycle-end> tag has a context-id of 4, which reflects the id value that is shown for <cycle-start>.

The items in this section of the log are explained as follows:

<cycle-start> and <cycle-end>
These tags represent a garbage collection cycle. Each tag has the following attributes:

type The type of garbage collection. This attribute can have the following values:

scavenge
Nursery collection is called a Scavenge.

global Mark-sweep garbage collection on the entire heap, with an optional Compact pass. For more information about global garbage collection, see: Detailed description of garbage collection

contextid
The contextid attribute of the <cycle-end> tag matches the id attribute of the corresponding <cycle-start> tag. In the example, the value of 4 indicates that this <cycle-end> tag corresponds to the <cycle-start id="4"> tag.

timestamp
The local timestamp at the time of the start or end of the garbage collection cycle.

intervalms
The amount of time, in milliseconds, since the start of the last collection of this type. For the <cycle-start> tag, this value therefore includes both the duration of the previous garbage collection cycle, and the interval between the end of the last collection cycle and the start of this collection cycle.

If you are using the balanced garbage collection policy, you might see the following line, which precedes the <cycle-start> tag:

```
<allocation-taxation id="28" taxation-threshold="2621440" timestamp="2014-02-17T16:21:44.325" intervalms="319.068">
</allocation-taxation>
```

This line indicates that the current garbage collection cycle was triggered due to meeting an allocation threshold that was set at the end of the previous cycle. The value of the threshold is reported.

Garbage collection increment:

A complete garbage collection increment is shown within <gc-start> and <gc-end> tags in the verbose output. Each garbage collection increment includes at least one garbage collection operation.

```
<gc-start id="5" type="scavenge" contextid="4" timestamp="2015-12-07T14:21:11.196">
  <mem-info id="6" free="3042472" total="3670016" percent="82">
    <mem type="nursery" free="0" total="524288" percent="0" />
    <mem type="tenure" free="3042472" total="3145728" percent="96">
      <mem type="soa" free="2885288" total="2988544" percent="96" />
```

```
      <mem type="loa" free="157184" total="157184" percent="100" />
    </mem>
    <remembered-set count="1852" />
  </mem-info>
</gc-start>
...
<gc-op id="7" type="scavenge" timems="3.107" contextid="4" timestamp="2015-12-07T14:21:11.199">
...
<gc-end id="8" type="scavenge" contextid="4" durationms="2.204" timestamp="2015-12-07T14:21:11.421">
  <mem-info id="9" free="3115152" total="3670016" percent="84">
    <mem type="nursery" free="72680" total="524288" percent="13" />
    <mem type="tenure" free="3042472" total="3145728" percent="96">
      <mem type="soa" free="2885288" total="2988544" percent="96" />
      <mem type="loa" free="157184" total="157184" percent="100" />
    </mem>
    <pending-finalizers system="1" default="0" reference="0" classloader="0" />
    <remembered-set count="1852" />
  </mem-info>
</gc-end>
```

The following details can be found in the log:

<gc-start>

This tag represents the start of a garbage collection increment. This tag has the following attributes:

type The type of garbage collection. This attribute can have the following values:

scavenge
Nursery collection is called a Scavenge.

global Mark-sweep garbage collection on the entire heap, with an optional Compact pass. For more information about global garbage collection, see: Detailed description of garbage collection

contextid
The `contextid` attribute matches the `id` attribute of the corresponding garbage collection cycle. In the example, the value of 4 indicates that this garbage collection increment is part of the garbage collection cycle that has the tag `<cycle-start id="4">`.

timestamp
The local time stamp at the start or end of the garbage collection increment.

The `<gc-start>` tag encloses a `<mem-info>` section, which provides information about the current state of the Java heap.

<gc-end>

This tag represents the end of a garbage collection increment. This tag has the following attributes:

type The type of garbage collection. This attribute can have the following values:

scavenge
Nursery collection is called a Scavenge.

global Mark-sweep garbage collection on the entire heap, with an optional Compact pass. For more information about global garbage collection, see: Detailed description of garbage collection

contextid
The contextid attribute matches the id attribute of the corresponding garbage collection cycle. In the example, the value of 4 indicates that this garbage collection increment is part of the garbage collection cycle that has the tag <cycle-start id="4">.

timestamp
The local time stamp at the start or end of the garbage collection increment.

usertimems
The total time in CPU seconds that the garbage collection threads spent in user mode.

systemtimems
The total time in CPU seconds that the garbage collection threads spent in kernel mode. A high systemtimems value can suggest that there is a high overhead in work sharing between garbage collection threads. If this is the case, you can use the **-Xgcthreads** option to lower the garbage collection thread count.

activeThreads
The number of active garbage collection threads during this garbage collection increment. This number might be lower than the number of garbage collection threads reported when garbage collection is initialized.

The <gc-end> tag encloses a <mem-info> section, which provides information about the current state of the Java heap.

<mem-info>
This tag shows the cumulative amount of free space and total space in the Java heap, calculated by summing the nursery heap size and the tenure heap size..

If you are using the Generational Concurrent Garbage Collector, the total value does not account for survivor space in the nursery. You can calculate the real total heap size by using the following formula:

```
reported-total-tenure-heap-size + reported-total-nursery-size/tilt-ratio
```

The tilt ratio is shown in the associated <gc-op> section of the garbage collection log.

<mem> Within each <mem-info> tag, multiple <mem> tags show the division of available memory across the various memory areas. Each <mem> tag shows the amount of free space and total space that is used in that memory area before and after a garbage collection event. The free space is shown as a figure and as a rounded-down percentage. The memory area is identified by the type attribute, which has one of the following values:

nursery
If you are using the Generational Concurrent Garbage Collector, nursery indicates that this <mem> tag applies to the new area of the Java heap. For more information about the Generational Concurrent Garbage Collector, see: Generational Concurrent Garbage Collector.

tenure Indicates that this <mem> tag applies to the tenure area, where objects are stored after they reach the tenure age. This memory type is further divided into soa and loa areas.

soa Indicates that this <mem> tag applies to the small object area of the tenure space. This area is used for the first allocation attempt for an object.

loa Indicates that this <mem> tag applies to the large object area of the tenure space. This area is used to satisfy allocations for large objects. For more information, see Large Object Area.

The <gc-end> tag also contains information about <pending-finalizers>. For more information and examples, see "Information about finalization" on page 122.

Related information:

Tilt ratio

Garbage collection operation:

Every garbage collection increment contains at least one garbage collection operation, which is shown in the verbose output with a <gc-op> tag.

The <gc-op> output contains subsections that describe operations that are specific to the type of garbage collection operation. These subsections might change from release to release, when improvements are made to the technology or when new data becomes available.

The following log excerpt shows an example of a garbage collection operation:

```
<gc-op id="7" type="scavenge" timems="1.127" contextid="4" timestamp="2010-11-23T00:41:32.515">
   ...
   ... subsections that are determined by the operation type
   ...
</gc-op>
```

The items in this section of the log are explained as follows:

<gc-op>

This tag represents a garbage collection operation, and has the following attributes:

type The type of garbage collection. The value of this attribute depends on the stage of the garbage collection cycle and the garbage collection policy that is in use. The value determines the subsections that appear within the <gc-op> tag, as described later in this topic.

Global garbage collection

The following type values can occur during any part of a global garbage collection cycle, and with the following garbage collection policies: generational concurrent (**gencon**), optimize for throughput (**optthruput**), and optimize for pause time (**optavgpause**).

mark The mark phase of garbage collection. During this phase, the garbage collector marks all the live objects. See "Subsections for mark operations" on page 119.

sweep The sweep phase of garbage collection. During this phase, the garbage collector identifies the unused parts of the heap, avoiding the marked objects. See "Subsections for sweep operations" on page 120.

compact
The compact phase of garbage collection. During this phase, the garbage collector moves objects to create larger, unfragmented areas of free memory. The garbage collector also changes references to moved objects to point to the new object location. Operations of this type might not occur because compaction is not always required. See "Subsections for compact operations" on page 120.

classunload
The garbage collector unloads classes and class loaders that have no live object instances. Operations of this type might not occur because class unloading is not always required. See "Subsections for classunload operations" on page 121.

Final stop-the-world part of a concurrent global garbage collection

The following `type` values can occur only in the final stop-the-world part of a concurrent global garbage collection cycle, and with the following garbage collection policies: **gencon** and **optavgpause**. These operations occur before mandatory mark-sweep operations, in the order shown.

tracing
The garbage collector traces and marks live objects before the final card-cleaning phase. Operations of this type occur only if the concurrent phase of the global garbage collection cycle is abnormally halted. Normally, tracing and marking is done concurrently.

rs-scan
The garbage collector traces and marks objects in the nursery that were found through the remembered set. The remembered set is a list of objects in the old (tenured) heap that have references to objects in the new area.

card-cleaning
The final card-cleaning phase before final stop-the-world marking. This phase is a normal step in incremental-update concurrent marking. This phase compensates for live object mutation during concurrent tracing and marking.

gencon garbage collection

The following `type` value applies only to the **gencon** garbage collection policy, and occurs during local and nursery collections.

scavenge
A scavenge operation involves tracing live nursery objects and moving them to the survivor area. The operation also includes fixing or adjusting object

references for the whole heap. See “Subsections for scavenge operations” on page 121.

timems The time, in milliseconds, taken to complete the garbage collection operation.

contextid
The `contextid` attribute matches the `id` attribute of the corresponding garbage collection cycle. In the example, the value of 4 indicates that this garbage collection increment is part of the garbage collection cycle that has the tag `<cycle-start id="4">`.

timestamp
The local time stamp at the time of the garbage collection operation.

The following information describes the subsections of the `<gc-op>` tag, which vary depending on the value of the `<gc-op>` **type** attribute.

Subsections for `mark` operations

The following log excerpt shows an example of a `mark` operation:

```
<gc-op id="9016" type="mark" timems="14.563" contextid="9013" timestamp="2015-09-28T14:47:49.927">
  <trace-info objectcount="1896901" scancount="1307167" scanbytes="34973672" />
  <finalization candidates="1074" enqueued="1" />
  <references type="soft" candidates="16960" cleared="10" enqueued="6" dynamicThreshold="12" maxThreshold="32" />
  <references type="weak" candidates="6514" cleared="1" enqueued="1" />
  <references type="phantom" candidates="92" cleared="0" enqueued="0" />
  <stringconstants candidates="18027" cleared="1" />
</gc-op>
```

The subsections within the `<gc-op>` tag are explained as follows:

<trace-info>
Contains general information about the objects traced. This tag has the following attributes:

objectcount
The number of objects discovered during the stop-the-world (STW) phase of marking.

scancount
The number of objects that are non-leaf objects: that is, they have at least one reference slot.

scanbytes
The total size in bytes of all scannable objects. (This is less than the total size of all live objects, the "live set.")

<finalization>
<references>
For information about the `<finalization>` and `<references>` elements, see “Information about finalization” on page 122 and “Information about reference processing” on page 123.

<stringconstants>
Contains general information about the objects traced. This tag has the following attributes:

candidates
The total number of string constants.

cleared
The number of string constants removed during this garbage collection cycle. (The number of string constants added since the previous global garbage collection is not explicitly reported.)

Subsections for sweep operations

The following log excerpt shows an example of a sweep operation:

```
<gc-op id="8979" type="sweep" timems="1.468" contextid="8974" timestamp="2015-09-28T14:47:49.141" />
```

There are no subsections within the <gc-op> tag.

Subsections for compact operations

The following log excerpt shows an example of a compact operation:

```
<gc-op id="8981" type="compact" timems="43.088" contextid="8974" timestamp="2015-09-28T14:47:49.184">
  <compact-info movecount="248853" movebytes="10614296" reason="compact on aggressive collection" />
</gc-op>
```

There is one subsection within the <gc-op> tag:

<compact-info>
This tag has the following attributes:

movecount
The number of objects moved.

movebytes
The size of the objects moved in bytes.

reason The reason for the compact operation:

compact to meet allocation
Unable to satisfy allocation even after mark-sweep.

compact on aggressive collection
Aggressive garbage collection is a global garbage collection that involves extra steps and gc-op operations to free as much memory as possible. One of those operations is compaction. Aggressive garbage collection might be triggered after a normal (non-agressive) Global garbage collection was unable to satisfy the allocate operation. Note that alternate Explicit garbage collections (for example, those invoked with **System.gc()**) are aggressive.

heap fragmented
Compaction to reduce fragmentation, as measured by internal metrics. There are a number of reasons to reduce fragmentation such as to prevent premature allocation failures with large objects, increase locality of objects and references, lower contention in allocation, or reduce frequency of Global garbage collections.

forced gc with compaction
A Global garbage collection that included a compact operation was explicitly requested, typically by using an agent or RAS tool, before a heap dump, for example.

low free space
An indication that free memory is less than 4%.

very low free space
An indication that free memory is less than 128 kB.

forced compaction
Compaction was explicitly requested with one of the JVM options such as **-Xcompactgc**.

compact to aid heap contraction
Objects were moved in the heap from high to low address ranges to create contiguous free space and thus aid heap contraction.

Subsections for `classunload` operations

The garbage collector unloads classes and class loaders that have no live object instances. Operations of this type might not occur because class unloading is not always required. The following log excerpt shows an example of a `classunload` operation:

```
<gc-op id="8978" type="classunload" timems="1.452" contextid="8974" timestamp="2015-09-28T14:47:49.140">
  <classunload-info classloadercandidates="1147" classloadersunloaded="3" classesunloaded="5"
                anonymousclassesunloaded="0" quiescems="0.000" setupms="1.408" scanms="0.041" postms="0.003" />
</gc-op>
```

There is one subsection within the `<gc-op>` tag:

<classunload-info>
This tag has the following attributes:

classloadercandidates
The total number of class loaders.

classloadersunloaded
The number of class loaders unloaded in this garbage collection cycle.

classesunloaded
The number of classes unloaded.

anonymousclassesunloaded
The number of anonymous classes unloaded. (Anonymous classes are unloaded individually and are reported separately.)

quiescems
setupms
scanms
postms The total time (in milliseconds) is broken down into four substeps.

Subsections for `scavenge` operations

Scavenge operations occur only with the **gencon** garbage collection policy. A scavenge operation runs when the allocate space within the nursery area is filled. During a scavenge, reachable objects are copied either into the survivor space within the nursery, or into the tenure space if they have reached the tenure age.

The following log excerpt shows an example of a scavenge operation:

```
<gc-op id="9029" type="scavenge" timems="2.723" contextid="9026" timestamp="2015-09-28T14:47:49.998">
  <scavenger-info tenureage="3" tenuremask="ffb8" tiltratio="89" />
  <memory-copied type="nursery" objects="11738" bytes="728224" bytesdiscarded="291776" />
  <memory-copied type="tenure" objects="6043" bytes="417920" bytesdiscarded="969872" />
```

```
  <finalization candidates="266" enqueued="0" />
  <references type="soft" candidates="94" cleared="0" enqueued="0" dynamicThreshold="12" maxThreshold="32" />
  <references type="weak" candidates="317" cleared="25" enqueued="17" />
</gc-op>
```

The subsections within the <gc-op> tag are explained as follows:

<scavenger-info>
Contains general information about the operation. This tag has the following attributes:

tenureage
The current age at which objects are promoted to the tenure area.

tenuremask

tiltratio
The tilt ratio (a percentage) after the last scavenge event and space adjustment. The scavenger redistributes memory between the allocate and survivor areas by using a process called "tilting". Tilting controls the relative sizes of the allocate and survivor spaces, and the tilt ratio is adjusted to maximize the amount of time between scavenges. A tilt ratio of 60% indicates that 60% of new space is reserved for allocate space and 40% for survivor space.

<memory-copied>
Indicates the quantity of object data that is flipped to the nursery area or promoted to the tenure area. This tag has the following attributes:

type One of the values `nursery` or `tenure`.

objects
The number of objects flipped to the nursery area or promoted to the tenure area.

bytes The number of bytes flipped to the nursery area or promoted to the tenure area.

bytesdiscarded
The number of bytes consumed in the nursery or tenure area but not successfully used for flipping or promotion. For each area, the total amount of consumed memory is the sum of the values of `bytes` and `bytesdiscarded`.

<finalization>
<references>
For information about the <finalization> and <references> elements, see “Information about finalization” and “Information about reference processing” on page 123.

Information about finalization:

The <finalization> section of the log records the number of enqueued finalizable objects that are in the current GC operation. The <pending-finalizers> section, which is found in the <gc-end> tag, records the current pending state. The current pending state is the sum of the enqueued finalizable objects from the current GC operation, plus all the objects from the past that are not yet finalized.

The following log excerpt shows an example of a <finalization> entry in the log:

```
<finalization candidates="1088" enqueued="10">
```

<finalization>
This tag shows the number of objects that contain finalizers and were queued for virtual machine finalization during the collection. This number is not equal to the number of finalizers that were run during the collection because finalizers are scheduled by the virtual machine. This tag has the following attributes:

candidates
Indicates the number of finalizable objects that were found in the GC cycle. The number includes live finalizable objects and those finalizable objects that are no longer alive since the last GC cycle. Only those objects that are no longer alive are enqueued for finalization.

enqueued
Indicates the fraction of candidates that are eligible for finalization.

The following log excerpt shows an example of a `<pending-finalizers>` entry in the log, which is recorded only in the `gc-end` section.

```
<pending-finalizers system="3" default="7" reference="40" classloader="0" />
```

<pending-finalizers>
Indicates the current state of queues of finalizable objects.

system Indicates the number of enqueued system objects.

default
Indicates the number of enqueued default objects. At the end of the GC cycle, the sum of `system` objects and `default` objects is larger than or equal to the fraction of candidates that are eligible for finalization from the same GC cycle.

reference
Indicates the number of enqueued references. That is, the number of references that were cleared and have a reference queue that is associated with them since the previous GC cycle. Typically, the number of pending references is larger or equal to the sum of enqueued weak, soft, and phantom references reported in the `gc-op` stanza of the same cycle.

classloader
Indicates the number of class loaders that are eligible for asynchronous unloading.

If the number of pending finalizers is larger than the number of candidates that are created by the current GC cycle, finalization cannot keep up with the influx. This situation might indicate suboptimal behavior. You can work out the number of outstanding pending finalizer objects at the beginning of a GC cycle by using the following calculation:

```
number of outstanding pending finalizer objects at the beginning =
number of pending finalizer objects at the end - candidates created in this cycle
```

Information about reference processing:

The `<references>` section in the verbose Garbage Collection (GC) logs contains information about reference processing.

The following log excerpt shows an example of a `<references>` entry in the log:

```
<references type="soft" candidates="16778" cleared="21" enqueued="14" dynamicThreshold="10" maxThreshold="32" />
<references type="weak" candidates="5916" cleared="33" enqueued="26" />
```

<references>

This tag provides information about Java reference objects, and has the following attributes:

type Indicates the type of the reference object. The type affects how the reference object is processed during garbage collection. The `type` attribute can have the following values:

soft Indicates that this object is an instance of the SoftReference class. Soft references are processed first during garbage collection.

weak Indicates that this object is an instance of the WeakReference class. Weak references are processed after soft references during garbage collection.

phantom
Indicates that this object is an instance of the PhantomReference class. Phantom references are processed after weak references during garbage collection.

candidates
Indicates the number of reference objects that were found in the GC cycle. The number includes reference objects whose referents are strong, soft, weak, or phantom reachable.

cleared
Indicates the number of reference objects that have a soft, weak, or phantom reachable referent, and that are cleared in this GC cycle.

enqueued
Indicates the fraction of cleared reference objects that are eligible for enqueuing. Eligible objects are cleared references objects that have a ReferenceQueue associated with them at reference creation time. The reference enqueuing is done by the finalization thread. For more information about the finalization section of the log, see "Information about finalization" on page 122.

dynamicThreshold
Applicable only to `soft` reference types. Indicates a dynamic value for the number of GC cycles (including local or global GC cycles) that a soft reference object can survive before it is cleared. The dynamic number is generated by internal heuristics that can reduce the threshold. For example, high heap occupancy might reduce the threshold from the maximum value.

maxThreshold
Applicable only to `soft` reference types. This value shows the maximum number of GC cycles (including local or global) that a soft reference object can survive before it is cleared.

Allocation failure:

Garbage collection cycles caused by an allocation failure are shown by `<af-start>` and `<af-end>` tags in the verbose output.

The `<af-start>` and `<af-end>` tags enclose the `<cycle-start>` and `<cycle-end>` tags. The `<af-start>` tag contains a `totalBytesRequested` attribute. This attribute specifies the number of bytes that were required by the allocations that caused this allocation failure. The `intervalms` attribute on the `af-start` tag is the time, in milliseconds, since the previous `<af-start>` tag. When the garbage collection cycle

caused by the allocation failure is complete, an `allocation-satisfied` tag is generated. This tag indicates that the allocation that caused the failure is now complete.

The following example shows two cycles within an af-start/af-end pair. Typically there is only one cycle, but in this example, Scavenge is not able to release enough memory in either Nursery or Tenure to meet the value of `totalBytesRequested`. This failure triggers global garbage collection, after which the allocation request is fulfilled.

```
<af-start id="2540" totalBytesRequested="8011256" timestamp="2015-12-07T14:00:34.657" intervalms="393.258" />
<cycle-start id="2541" type="scavenge" contextid="0" timestamp="2015-12-07T14:00:34.657" intervalms="393.269" />
...
  <gc-end id="2545" type="scavenge" contextid="2541" durationms="5.163" usertimems="25.996" systemtimems="3.999" timestamp="2015-12-07T14:00:34.662" activeThreads="24">
    <mem-info id="2546" free="9330208" total="111149056" percent="8">
    <mem type="nursery" free="1484208" total="27787264" percent="5">
...
<cycle-end id="2547" type="scavenge" contextid="2541" timestamp="2015-12-07T14:00:34.662" />
<cycle-start id="2548" type="global" contextid="0" timestamp="2015-12-07T14:00:34.662" intervalms="7196.493" />
...
<cycle-end id="2556" type="global" contextid="2548" timestamp="2015-12-07T14:00:34.668" />
<allocation-satisfied id="2557" threadId="00000000022BCF00" bytesRequested="8011256" />
<af-end id="2558" timestamp="2015-12-07T14:00:34.671" />
```

The items in this section of the log are explained as follows:

<af-start> and <af-end>
: This tag is generated when an allocation failure occurs, and contains a garbage collection cycle, indicated by the `<cycle-start>` and `<cycle-end>` tags. This tag has the following attributes:

  totalBytesRequested
  : The number of bytes that were required by the allocations that caused this allocation failure.

  timestamp
  : The local timestamp at the time of the allocation failure.

  intervalms
  : The time, in milliseconds, since the previous `<af-start>` tag was generated.

<allocation-satisfied>
: This tag indicates that the allocation that caused the failure is complete. This tag is generated when the garbage collection cycle that was caused by the allocation failure is complete. This tag has the following attributes:

  thread The Java thread identifier that triggers garbage collection.

  bytesRequested
  : This attribute is identical to the totalBytesRequested attribute that is seen in the `<af-start>` tag.

Tracing problems with Balanced garbage collection

You can trace problems with garbage collection using the **-Xtgc** options. Some of these options are changed when using the Balanced Garbage Collection policy.

The following **-Xtgc** options are no longer accepted when used with **-Xgcpolicy:balanced**:

- `concurrent`
- `references`
- `scavenger`

In all cases, the JVM fails to start.

The following **-Xtgc** options can be used with **-Xgcpolicy:balanced**:

- backtrace
- compaction
- dump
- freeList
- parallel
- terse

For further information about the **-Xtgc** options, see Tracing garbage collection operations.

Using the JVMTI

The Java Virtual Machine Tool Interface (JVMTI) is a two-way interface that allows a native agent to analyze a Java virtual machine (JVM).

The JVMTI is a standard interface from Oracle that allows third parties to develop debugging, profiling, and monitoring tools for a JVM. The interface allows an agent to request information or to trigger a function within a JVM, and to receive notifications. Several agents can be attached to a JVM at any one time. For more information about using JVMTI, see the Java 6 Diagnostics Guide, http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/com.ibm.java.doc.diagnostics.60/diag/tools/jvmti.html.

IBM JVMTI extensions

The IBM SDK provides extensions to JVMTI that enhance the diagnostic capabilities of this interface. New extensions are available.

New IBM JVMTI extensions are available for the following tasks:

- Modifying the logging configuration of the JVM.
- Querying native memory usage.
- Finding and removing shared class caches.
- Subscribing to, and unsubscribing from, verbose garbage collection logging.

IBM JVMTI extensions - API reference:

Reference information for the IBM SDK extensions to the JVMTI.

Use the information in this section to query or control J9 VM functions by using the JVMTI interface.

Reference material for IBM JVMTI extensions that are included with the IBM SDK, Java Technology Edition, Version 6 can be found here: ../../../../com.ibm.java.doc.diagnostics.60/diag/tools/jvmti_extensions_ref.html.

Querying runtime environment native memory categories:

You can query the total native memory consumption of the runtime environment for each memory category using the GetMemoryCategories() API.

The GetMemoryCategories() API has the JVMTI Extension Function identifier com.ibm.GetMemoryCategories. The identifier is declared as macro COM_IBM_GET_MEMORY_CATEGORIES in `ibmjvmti.h`.

Native memory is memory requested from the operating system using library functions such as malloc() and mmap(). Runtime environment native memory use

is grouped under high-level memory categories, as described in the Javadump section "Native memory (NATIVEMEMINFO)" on page 72. The data returned by the GetMemoryCategories() API is consistent with this format.

```
jvmtiError GetMemoryCategories(jvmtiEnv* env, jint version, jint max_categories,
jvmtiMemoryCategory * categories_buffer, jint * written_count_ptr, jint *
total_categories_ptr);
```

The extension writes native memory information to a memory buffer specified by the user. Each memory category is recorded as a jvmtiMemoryCategory structure, whose format is defined in `ibmjvmti.h`.

You can use the GetMemoryCategories() API to work out the buffer size you must allocate to hold all memory categories defined inside the JVM. To calculate the size, call the API with a NULL **categories_buffer** argument and a non-NULL **total_categories_ptr** argument.

Parameters:

env: A pointer to the JVMTI environment.

version: The version of the jvmtiMemoryCategory structure that you are using. Use COM_IBM_GET_MEMORY_CATEGORIES_VERSION_1 for this argument, unless you must work with an obsolete version of the jvmtiMemoryCategory structure.

max_categories: The number of jvmtiMemoryCategory structures that can fit in **categories_buffer**.

categories_buffer: A pointer to the memory buffer for holding the result of the GetMemoryCategories() call. The number of jvmtiMemoryCategory slots available in **categories_buffer** must be accurately specified with **max_categories**, otherwise GetMemoryCategories() can overflow the memory buffer. The value can be NULL.

written_count_ptr: A pointer to *jint* to store the number of jvmtiMemoryCategory structures to be written to **categories_buffer**. The value can be NULL.

total_categories_ptr: A pointer to *jint* to store the total number of memory categories declared in the JVM. The value can be NULL.

Returns:

`JVMTI_ERROR_NONE`: Success.

`JVMTI_ERROR_UNSUPPORTED_VERSION`: Unrecognized value passed for **version**.

`JVMTI_ERROR_ILLEGAL_ARGUMENT`: Illegal argument; **categories_buffer**, **count_ptr** and **total_categories_ptr** all have NULL values.

`JVMTI_ERROR_INVALID_ENVIRONMENT`: The **env** parameter is invalid.

`JVMTI_ERROR_OUT_OF_MEMORY`: Memory category data is truncated because **max_categories** is not large enough.

Querying JVM log options:

You can query the JVM log options that are set using the QueryVmLogOptions() API.

The QueryVmLogOptions() API has the JVMTI Extension Function identifier com.ibm.QueryVmLogOptions. The identifier is declared as macro COM_IBM_QUERY_VM_LOG_OPTIONS in `ibmjvmti.h`.

To query the current JVM log options, use:

```
jvmtiError QueryVmLogOptions(jvmtiEnv* jvmti_env, jint buffer_size, \
void* options, jint* data_size_ptr)
```

This extension returns the current log options as an ASCII string. The syntax of the string is the same as the **-Xlog** command-line option, with the initial **-Xlog:** omitted. For example, the string `"error,warn"` indicates that the JVM is set to log error and warning messages only. For more information about using the **-Xlog** option, see "-Xlog" on page 145. If the memory buffer is too small to contain the current JVM log option string, 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 string.

Returns:

`JVMTI_ERROR_NONE`: Success

`JVMTI_ERROR_NULL_POINTER`: The **options** or **data_size_ptr** parameters are null.

`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 supplied memory buffer is too small.

Setting JVM log options:

You can set the log options for a JVM using the same syntax as the **-Xlog** command-line option.

The SetVmLogOptions() API has the JVMTI Extension Function identifier com.ibm.SetVmLogOptions. The identifier is declared as macro COM_IBM_SET_VM_LOG_OPTIONS in `ibmjvmti.h`.

To set the JVM log options use:

```
jvmtiError SetVmLogOptions(jvmtiEnv* jvmti_env, char* options_buffer)
```

The log option is passed in as an ASCII character string. Use the same syntax as the **-Xlog** command-line option, with the initial **-Xlog:** omitted. For example, to set the JVM to log error and warning messages, pass in a string containing `"error,warn"`. For more information about using the **-Xlog** option, see "-Xlog" on page 145.

Parameters:

jvmti_env: A pointer to the JVMTI environment.

options_buffer: A pointer to memory containing the log option.

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_ILLEGAL_ARGUMENT: The parameter **option** contains an invalid **-Xlog** string.

Finding shared class caches:

You can search for caches by using the IterateSharedCaches() API.

IterateSharedCaches()

The IterateSharedCaches() API has the JVMTI Extension Function identifier com.ibm.IterateSharedCaches. The identifier is declared as macro COM_IBM_ITERATE_SHARED_CACHES in ibmjvmti.h.

To search for shared class caches that exist in a specified cache directory, use:

```
jvmtiError IterateSharedCaches(jvmtiEnv* env, jint version, const char *cacheDir,
jint flags, jboolean useCommandLineValues, jvmtiIterateSharedCachesCallback
callback, void *user_data);
```

This extension searches for shared class caches in a specified directory. Information about the caches is returned in a structure that is populated by a user specified callback function. You can specify the search directory by either:

- Setting the value of **useCommandLineValues** to true and specifying the directory on the command line. If you do not specify a directory on the command line, the default platform location is used.
- Setting the value of **useCommandLineValues** to false and using the **cacheDir** parameter. To accept the default platform location, specify **cacheDir** with a NULL value.

Parameters:

env: A pointer to the JVMTI environment.

version: Version information for IterateSharedCaches, which describes the jvmtiSharedCacheInfo structure passed to the jvmtiIterateSharedCachesCallback function. The only value permitted is COM_IBM_ITERATE_SHARED_CACHES_VERSION_1.

cacheDir: When the value of **useCommandLineValues** is false, specify the absolute path of the directory for the shared class cache. If the value is null, the platform-dependent default is used.

flags: Reserved for future use. The only value permitted is COM_IBM_ITERATE_SHARED_CACHES_NO_FLAGS.

useCommandLineValues: Set this value to true when you want to specify the cache directory on the command line. Set this value to false when you want to use the **cacheDir** parameter.

callback: A function pointer to a user provided callback routine jvmtiIterateSharedCachesCallback.

user_data: User supplied data, passed as an argument to the callback function.

```
jint (JNICALL *jvmtiIterateSharedCachesCallback)(jvmtiEnv *env,jvmtiSharedCacheInfo
*cache_info, void *user_data);
```

Returns:

JVMTI_ERROR_NONE: Success.

JVMTI_ERROR_OUT_OF_MEMORY: There is insufficient system memory to process the request.

JVMTI_ERROR_INVALID_ENVIRONMENT: The **env** parameter is not valid.

JVMTI_ERROR_WRONG_PHASE: The extension has been called outside the JVMTI live phase.

JVMTI_ERROR_UNSUPPORTED_VERSION: The **version** parameter is not valid.

JVMTI_ERROR_NULL_POINTER: The **callback** parameter is NULL.

JVMTI_ERROR_NOT_AVAILABLE: The shared classes feature is not enabled in the JVM.

JVMTI_ERROR_ILLEGAL_ARGUMENT: The **flags** parameter is not valid.

JVMTI_ERROR_INTERNAL: This error is returned when the jvmtiIterateSharedCachesCallback returns JNI_ERR.

jvmtiIterateSharedCachesCallback function

The jvmtiIterateSharedCachesCallback function is called with the following parameters:

Parameters:

env: A pointer to the JVMTI environment when calling COM_IBM_ITERATE_SHARED_CACHES.

cache_info: A jvmtiSharedCacheInfo structure containing information about a shared cache.

user_data: User supplied data, passed as an argument to IterateSharedCaches.

The following values are returned by the jvmtiIterateSharedCachesCallback function.

Returns:

JNI_OK: Continue iterating.

JNI_ERR: Stop iterating, which causes IterateSharedCaches to return JVMTI_ERROR_INTERNAL

jvmtiSharedCacheInfo structure

The structure of jvmtiSharedCacheInfo:

```
typedef struct jvmtiSharedCacheInfo {
const char *name; - the name of the shared cache
jboolean isCompatible; - if the shared cache is compatible with this JVM
jboolean isPersistent; - true if the shared cache is persistent, false if its non-
                         persistent
jint os_shmid; - the OS shared memory ID associated with a non-persistent cache,
                 -1 otherwise
jint os_semid; - the OS shared semaphore ID associated with a non-persistent cache,
                 -1 otherwise
```

```
jint modLevel; - one of COM_IBM_SHARED_CACHE_MODLEVEL_JAVA5,
                 COM_IBM_SHARED_CACHE_MODLEVEL_JAVA6,
                 COM_IBM_SHARED_CACHE_MODLEVEL_JAVA7
jint addrMode; - one of COM_IBM_SHARED_CACHE_ADDRMODE_32,
                 COM_IBM_SHARED_CACHE_ADDRMODE_64
jboolean isCorrupt; - if the cache is corrupted
jlong cacheSize; - the total usable shared class cache size, or -1 when
                   isCompatible is false
jlong freeBytes; - the amount of free bytes in the shared class cache, or -1 when
                   isCompatible is false
jlong lastDetach; - the last detach tiime specified in milliseconds since
                    00:00:00 on January 1, 1970 UTC.
} jvmtiSharedCacheInfo;
```

For information about the equivalent Java APIs, see "Utility APIs" on page 105.

Removing a shared class cache:

You can remove a shared class cache using the DestroySharedCache() API.

The DestroySharedCache() API has the JVMTI Extension Function identifier com.ibm.DestroySharedCache. The identifier is declared as macro COM_IBM_DESTROY_SHARED_CACHE in `ibmjvmti.h`.

To remove a shared cache, use:

```
jvmtiError DestroySharedCache(jvmtiEnv *env, const char *cacheDir, const char *name,
jint persistence, jboolean useCommandLineValues, jint *internalErrorCode);
```

This extension removes a named shared class cache of a given persistence type, in a given directory. You can specify the cache name, persistence type, and directory by either:

- Setting **useCommandLineValues** to `true` and specifying the values on the command line. If a value is not available, the default values for the platform are used.
- Setting **useCommandLineValues** to `false` and using the **cacheDir**, **persistence** and **cacheName** parameters to identify the cache to be removed. To accept the default value for **cacheDir** or **cacheName**, specify the parameter with a NULL value.

Parameters:

env: A pointer to the JVMTI environment.

cacheDir: When the value of **useCommandLineValues** is `false`, specify the absolute path of the directory for the shared class cache. If the value is NULL, the platform-dependent default is used.

cacheName: When the value of **useCommandLineValues** is `false`, specify the name of the cache to be removed. If the value is NULL, the platform-dependent default is used.

persistence: When the value of **useCommandLineValues** is `false`, specify the type of cache to remove. This parameter must have one of the following values:

- `PERSISTENCE_DEFAULT`: The default value for the platform.
- `PERSISTENT`.
- `NONPERSISTENT`.

useCommandLineValues: Set this value to `true` when you want to specify the shared class cache name, persistence type, and directory on the command

line. Set this value to false when you want to use the **cacheDir**, **persistence** and **cacheName** parameters instead.

internalErrorCode: If not NULL, this value is set to one of the following constants when JVMTI_ERROR_INTERNAL is returned.

- COM_IBM_DESTROYED_NONE: Set when the function fails to remove any caches.
- COM_IBM_DESTROY_FAILED_CURRENT_GEN_CACHE: Set when the function fails to remove the existing current generation cache, irrespective of the state of older generation caches.
- COM_IBM_DESTROY_FAILED_OLDER_GEN_CACHE: Set when the function fails to remove any older generation caches. The current generation cache does not exist or is successfully removed

This value is set to COM_IBM_DESTROYED_ALL_CACHE when JVMTI_ERROR_NONE is returned.

Returns:

JVMTI_ERROR_NONE: Success. No cache exists or all existing caches of all generations are removed.

JVMTI_ERROR_OUT_OF_MEMORY: There is insufficient system memory to process the request.

JVMTI_ERROR_INVALID_ENVIRONMENT: The **env** parameter is not valid.

JVMTI_ERROR_WRONG_PHASE: The extension has been called outside the JVMTI live phase.

JVMTI_ERROR_NOT_AVAILABLE: The shared classes feature is not enabled in the JVM.

JVMTI_ERROR_ILLEGAL_ARGUMENT: The **persistence** parameter is not valid.

JVMTI_ERROR_INTERNAL: Failed to remove any existing cache with the given name. See the value of **internalErrorCode** for more information about the failure.

For information about the equivalent Java APIs, see "Utility APIs" on page 105.

Subscribing to verbose garbage collection logging:

You can subscribe to verbose Garbage Collection (GC) data logging through an IBM JVMTI extension.

The RegisterVerboseGCSubscriber() API has the JVMTI Extension function identifier com.ibm.RegisterVerboseGCSubscriber. The identifier is declared as macro COM_IBM_REGISTER_VERBOSEGC_SUBSCRIBER in `ibmjvmti.h`.

To register a subscription to verbose GC data logging, use:

```
jvmtiError RegisterVerboseGCSubscriber(jvmtiEnv* jvmti_env, char *description,
jvmtiVerboseGCSubscriber subscriber, jvmtiVerboseGCAlarm alarm, void
```

An ASCII character string describing the subscriber must be passed in.

An arbitrary pointer to user data must be supplied. This pointer is passed to the subscriber and alarm functions each time these functions are called. This pointer can be NULL.

A pointer to a subscription ID must be supplied. This pointer is returned by the RegisterVerboseGCSubscriber call if successful. The value must be supplied to a future call to DeregisterVerboseGCSubscriber.

Parameters:

jvmti_env: A pointer to the JVMTI environment.

description: A string that describes your subscriber.

subscriber: A function of type jvmtiVerboseGCSubscriber.

alarm: A function pointer of type jvmtiVerboseGCAlarm.

user_data: User data that is passed to the subscriber function.

subscription_id: A pointer to a subscription identifier that is returned.

Returns:

JVMTI_ERROR_NONE: Success.

JVMTI_ERROR_NULL_POINTER: One of the supplied parameters 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 not valid.

JVMTI_ERROR_WRONG_PHASE: The extension has been called outside the JVMTI live phase.

JVMTI_ERROR_NOT_AVAILABLE: GC verbose logging is not available.

JVMTI_ERROR_INTERNAL: An internal error has occurred.

The subscriber function type

The jvmtiVerboseGCSubscriber function is called with the following parameters:

```
typedef jvmtiError (*jvmtiVerboseGCSubscriber)(jvmtiEnv *jvmti_env, const char
*record, jlong length, void *user_data);
```

The subscriber function must be of type jvmtiVerboseGCSubscriber, which is declared in ibmjvmti.h. This function is called with each record of verbose logging data produced by the JVM. The verbose logging record supplied to the subscriber function is valid only for the duration of the function. If the subscriber wants to save the data, the data must be copied elsewhere. If the subscriber function returns an error, the alarm function is called, and the subscription is de-registered.

Alarm function parameters:

jvmti_env: A pointer to the JVMTI environment.

record: An ascii string that contains a verbose log record.

length: The number of ascii characters in the verbose log record.

user_data: User data supplied when the subscriber is registered.

The alarm function type

The jvmtiVerboseGCAlarm function is called with the following parameters:

```
typedef jvmtiError (*jvmtiVerboseGCAlarm)(jvmtiEnv *jvmti_env, void
*subscription_id, void *user_data);
```

The alarm function must be of type jvmtiVerboseGCAlarm, which is declared in ibmjvmti.h. This function is called if the subscriber function returns an error.

Alarm function parameters:

jvmti_env: A pointer to the JVMTI environment.

user_data: User data supplied when the subscriber is registered.

subscription_id: The subscription identifier.

Unsubscribing from verbose garbage collection logging:

You can unsubscribe from verbose Garbage Collection (GC) data logging through an IBM JVMTI extension.

The DeregisterVerboseGCSubscriber() API has the JVMTI Extension Function identifier com.ibm.DeregisterVerboseGCSubscriber. The identifier is declared as macro COM_IBM_DEREGISTER_VERBOSEGC_SUBSCRIBER in `ibmjvmti.h`.

To unsubscribe from verbose GC data logging, use:

```
jvmtiError DeregisterVerboseGCSubscriber(jvmtiEnv* jvmti_env, void *userData,
void *subscription_id)
```

You must supply the subscription ID returned by the call to RegisterVerboseGCSubscriber. The previously registered subscriber function is no longer called with future verbose logging records.

Parameters:

jvmti_env: A pointer to the JVMTI environment.

subscription_id: The subscription identifier.

Returns:

`JVMTI_ERROR_NONE`: Success.

`JVMTI_ERROR_NULL_POINTER`: The **subscription_id** 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 not valid.

`JVMTI_ERROR_WRONG_PHASE`: The extension has been called outside the JVMTI live phase.

Using the DTFJ interface

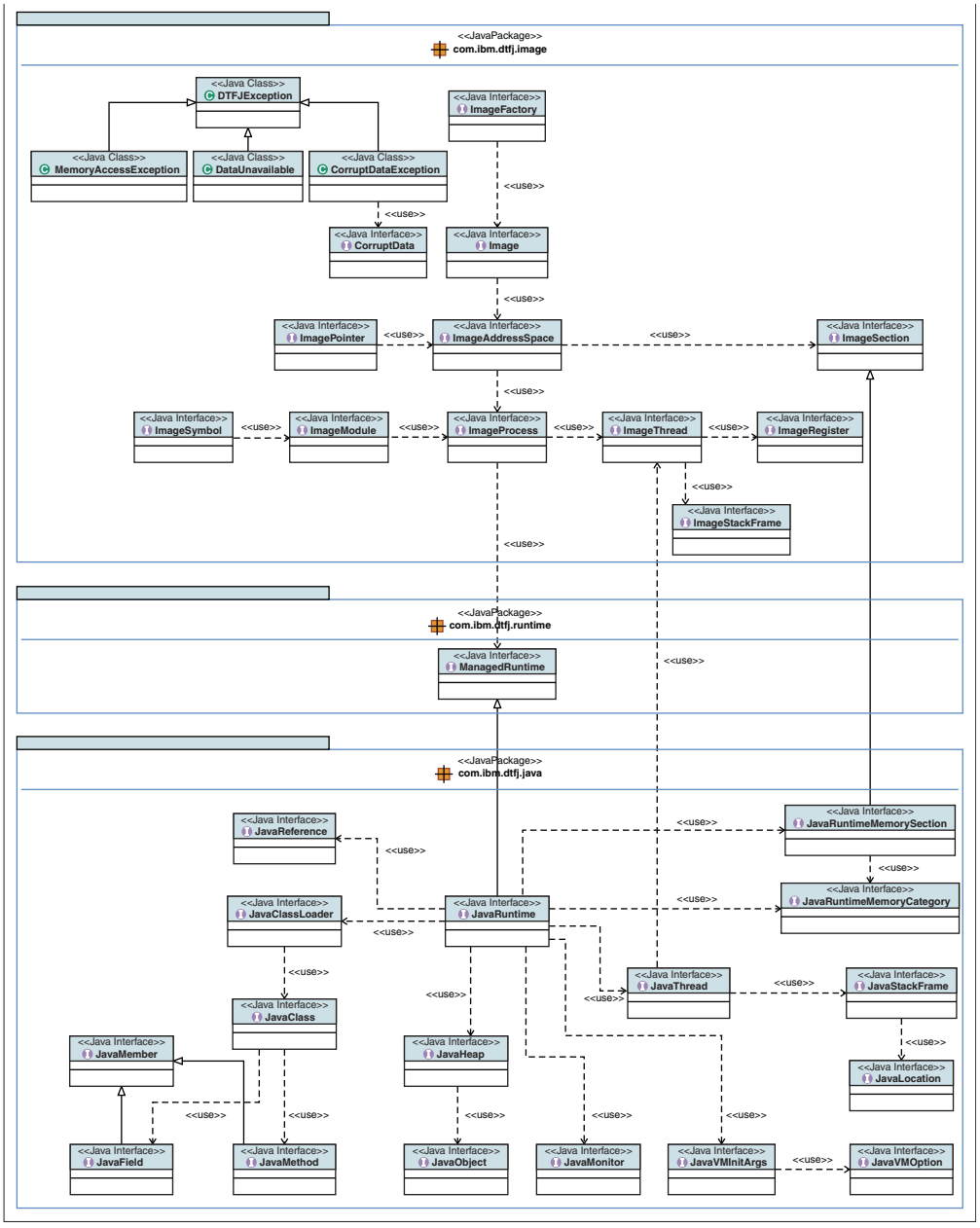
The Diagnostic Tool Framework for Java (DTFJ) interface has been updated. You can now write applications that obtain information about native memory from a system dump or javadump.

To create applications that use DTFJ, you must use the DTFJ interface. The DTFJ API has been enhanced to enable you to obtain information about native memory. Native memory is memory requested from the operating system using library functions such as malloc() and mmap(). When the runtime environment allocates native memory, the memory is associated with a high-level memory category. Each memory category has two running counters:

- The total number of bytes allocated but not yet freed.
- The number of native memory allocations that have not been freed.

Each memory category can have subcategories.

The following diagram illustrates the DTFJ interface:



For more information about using the DTFJ interface with a system dump or javadump, see ../../../../com.ibm.java.doc.diagnostics.60/diag/tools/dtfj_overview.html.

Chapter 12. Reference

This reference information applies only to IBM SDK, Java Technology Edition, Version 6 (J9 VM 2.6).

General reference information for IBM SDK, Java Technology Edition, Version 6 can be found in the User Guides and Diagnostic Guide that are available in the IBM Information Center: http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/welcome/welcome_javasdk_version.html.

Command-line options

There are a number of command-line options that you can use with the runtime environment.

These options supplement the command-line options documented in the IBM SDK, Java Technology Edition, Version 6 guides.

This chapter provides the following information:

System property command-line options

Use the system property command-line options to set up your system.

-D<*name*>=<*value*>
Sets a system property.

-Dcom.ibm.CORBA.Debug.Component

This system property can be used with **-Dcom.ibm.CORBA.Debug=true** to generate trace output for specific Object Request Broker (ORB) subcomponents such as MARSHAL or DISPATCH. This finer level of tracing helps you debug problems with ORB operations.

-Dcom.ibm.CORBA.Debug.Component=name
Where `name` can be one of the following ORB subcomponents:

- `DISPATCH`
- `MARSHAL`
- `TRANSPORT`
- `CLASSLOADER`
- `ALL`

When you want to trace more than one of these subcomponents, each subcomponent must be separated by a comma. The default value is `ALL`.

Note: This option has no effect unless it is used with the system property **-Dcom.ibm.CORBA.Debug=true**.

The following setting enables tracing for the DISPATCH, TRANSPORT, and CLASSLOADER components:

```
-Dcom.ibm.CORBA.Debug=true -Dcom.ibm.CORBA.Debug.Component=DISPATCH,TRANSPORT,CLASSLOADER
```

-Dcom.ibm.UseCLDR16

Use the **-Dcom.ibm.UseCLDR16** system property to change the default locale translation files used.

Purpose

From service refresh 1, changes are made to the locale translation files to make them consistent with Oracle JDK 6. To understand the differences in detail, see http://www.ibm.com/support/docview.wss?uid=swg21568667. Include the **-Dcom.ibm.UseCLDR16** system property on the command-line to revert to the locale translation files used in earlier releases.

-Dcom.ibm.xtq.processor.overrideSecureProcessing

This system property affects the XSLT processing of extension functions or extension elements when Java security is enabled.

Purpose

From service refresh 6, the use of extension functions or extension elements is not allowed when Java security is enabled. This change is introduced to enhance security. This system property can be used to revert to the behavior in earlier releases.

Parameters

com.ibm.xtq.processor.overrideSecureProcessing=*true*
To revert to the behavior in earlier releases of the IBM SDK, set this system property to *true*.

-Dibm.disableAltProcessor

This option stops the ALT-key, when pressed, from highlighting the first menu in the active window of the user interface.

-Dibm.disableAltProcessor=true
Set this property on the command line to prevent the ALT-key from highlighting the first menu in the active window.

Note: If your application uses a Windows Look and Feel (com.sun.java.swing.plaf.windows.WindowsLookAndFeel), this option has no effect.

-Djava.util.Arrays.useLegacyMergeSort

Changes the implementation of java.util.Collections.sort(list, comparator) in this release.

The Java SE 6 implementation of java.util.Collections.sort(list, comparator) relies on the Comparator function, which implements the conditions greater than, less than, and equal. However, the Java SE 5.0 implementation of java.util.Collections.sort(list, comparator) can accept the Comparator function, which implements only the conditions greater than and less than. From IBM SDK, Java Technology Edition, Version 6 (J9 VM 2.6) service refresh 8 fix pack 1 onwards, you can switch between the Java SE 5.0 and Java SE 6 implementation.

-Djava.util.Arrays.useLegacyMergeSort=[true | false]
Setting the value to `true` changes the Comparator function to the Java SE 5.0 implementation. The default for this setting is `false`.

JVM command-line options

This reference section provides a list of command-line options that are new, or changed, when IBM SDK, Java Technology Edition, Version 6 uses an IBM J9 2.6 virtual machine.

To see a complete list of JVM command-line options, see the Java 6 Diagnostics Guide, ../../../com.ibm.java.doc.diagnostics.60/diag/appendixes/cmdline/commands_jvm.html. Use these options to configure your JVM. The options prefixed with **-X** are nonstandard.

Conventions

Options shown with values that are in braces signify that one of the values must be chosen. For example:

-Xverify:{remote | all | none}

Options shown with values that are in brackets signify that the values are optional. For example:

-Xrunhprof[:**help**][<*suboption*>=<*value*>...]

-XCEEHDLR (31-bit z/OS only)

The `-XCEEHDLR` option is used to control 31-bit z/OS JVM Language Environment condition handling.

The `-XCEEHDLR` option is available on the 31-bit z/OS JVM. Use the `-XCEEHDLR` option if you want the new behavior for the Java and COBOL interoperability batch mode environment, because this option makes signal and condition handling behavior more predictable in a mixed Java and COBOL environment.

When the `-XCEEHDLR` option is enabled, a condition triggered by an arithmetic operation while executing a Java Native Interface (JNI) component causes the JVM to convert the Language Environment condition into a Java ConditionException.

When the `-XCEEHDLR` option is used

The JVM does not install POSIX signal handlers for the following signals:

- SIGBUS
- SIGFPE
- SIGILL
- SIGSEGV
- SIGTRAP

Instead, user condition handlers are registered by the JVM, using the CEEHDLR() method. These condition handlers are registered every time a thread calls into the JVM. Threads call into the JVM using the Java Native Interface and including the invocation interfaces, for example JNI_CreateJavaVM.

The runtime environment continues to register POSIX signal handlers for the following signals:

- SIGABRT
- SIGINT
- SIGQUIT
- SIGTERM

Signal chaining using the `libjsig.so` library is not supported.

Behavior of JVM condition handlers when the `-XCEEHDLR` option is used

Condition handler actions take place in the following sequence:

1. All severity 0 and severity 1 conditions are percolated.
2. If a Language Environment condition is triggered in JNI code as a result of an arithmetic operation, the JVM condition handler resumes executing Java code as if the JNI native code had thrown a com.ibm.le.conditionhandling.ConditionException exception. This exception class is a subclass of java.lang.RuntimeException.

   Note: The Language Environment conditions that correspond to arithmetic operations are CEE3208S through CEE3234S. However, the Language Environment does not deliver conditions CEE3208S, CEE3213S, or CEE3234S to C applications, so the JVM condition handler will not receive them.
3. If the condition handling reaches this step, the condition is considered to be unrecoverable. RAS diagnostic information is generated, and the JVM ends by calling the CEE3AB2() service with abend code 3565, reason code 0, and cleanup code 0.

-Xcheck

Use the **-Xcheck** option to check critical JVM functions.

Purpose

The **-Xcheck** option checks JVM functions, including the class loader, garbage collector, JNI function, and memory. The syntax for this option is **-Xcheck**[:<option>].

Parameters

-Xcheck:classpath
Displays a warning message if an error is discovered in the class path; for example, a missing directory or JAR file.

-Xcheck:dump
Runs checks on AIX and Linux operating system settings during JVM startup. Messages are issued if the operating system has dump options or limits set that might truncate system dumps.

Note: Not supported on Windows or z/OS.

On AIX, the following messages are possible:

JVMJ9VM133W The system core size hard ulimit is set to <value>,system dumps may be truncated
This message indicates that the AIX operating system user limit is set to restrict the size of system dumps to the value indicated. If a system dump is produced by the JVM it might be truncated, and therefore of

greatly reduced value in investigating the cause of crashes and other issues. For more information on how to set user limits on AIX, see Enabling full AIX core files.

JVMJ9VM134W The system fullcore option is set to FALSE, system dumps may be truncated

This message indicates that the AIX operating system `Enable full CORE dump` option is set to *FALSE*. This setting might result in truncated system dumps. For more information about how to set this option correctly on AIX, see Enabling full AIX core files.

On Linux, the following messages are possible:

JVMJ9VM133W The system core size hard ulimit is set to <value>, system dumps may be truncated.

This message indicates that the Linux operating system user limit is set to restrict the size of system dumps to the value indicated. If a system dump is produced by the JVM, it might be truncated and therefore of greatly reduced value in investigating the cause of crashes and other issues. Review the documentation that is provided for your operating system to correctly configure the value for `ulimits`. For further information, see Setting up and checking your Linux environment.

JVMJ9VM135W /proc/sys/kernel/core_pattern setting "|/usr/libexec/abrt-hook-ccpp %s %c %p %u %g %t e" specifies that core dumps are to be piped to an external program. The JVM may be unable to locate core dumps and rename them.

This message means that an external program, **abrt-hook-ccpp**, is configured in the operating system to intercept any system dump files that are generated. This program is part of the Automatic Bug Reporting Tool (ABRT). For more information, see Automatic Bug Reporting Tool. This tool might interfere with the JVM's system dump file processing by renaming or truncating system dumps. Review the configuration of the ABRT tool and messages that are written by the tool in `/var/log/messages`. If problems occur when generating system dumps from the JVM, consider disabling ABRT.

JVMJ9VM135W /proc/sys/kernel/core_pattern setting "|/usr/share/apport/apport %p %s %c" specifies that core dumps are to be piped to an external program. The JVM may be unable to locate core dumps and rename them.

This message means that an external program, **apport**, is configured in the operating system to intercept any system dump files that are generated. For more information about this tool, see: Apport The tool might interfere with the JVM's system dump file processing by renaming or truncating system dumps. Review the configuration of the Apport tool and messages that are written by the tool in `/var/log/apport.log`. If problems occur when generating system dumps from the JVM, consider disabling the Apport tool.

JVMJ9VM136W "/proc/sys/kernel/core_pattern setting "/tmp/cores/core.%e.%p.%h.%t " specifies a format string for renaming core dumps. The JVM may be unable to locate core dumps and rename them.

This message indicates that the Linux `/proc/sys/kernel/core_pattern` option is set to rename system dumps. The tokens that are used in the operating system dump name might interfere with the JVM's system dump file processing, in particular with file names specified in the JVM **-Xdump** options. If problems occur when generating system dumps from the JVM, consider changing the `/proc/sys/kernel/core_pattern` setting to the default value of `core`.

-Xcheck:gc[:<*scan options*>][:<*verify options*>][:<*misc options*>]
Runs checks on garbage collection. By default, no checks are done. See the output of **-Xcheck:gc:help** for more information.

-Xcheck:jni[**:help**][:<*option*>=<*value*>]
Runs additional checks for JNI functions. This option is equivalent to **-Xrunjnichk**. By default, no checks are done.

-Xcheck:memory[:<*option*>]
Identifies memory leaks inside the JVM using strict checks that cause the JVM to exit on failure. If no option is specified, **all** is used by default. The available options are as follows:

all
Enables checking of all allocated and freed blocks on every free and allocate call. This check of the heap is the most thorough. It typically causes the JVM to exit on nearly all memory-related problems soon after they are caused. This option has the greatest affect on performance.

callsite=<*number of allocations*>
Displays callsite information every <*number of allocations*>. De-allocations are not counted. Callsite information is presented in a table with separate information for each callsite. Statistics include:

- The number and size of allocation and free requests since the last report.
- The number of the allocation request responsible for the largest allocation from each site.

Callsites are presented as `sourcefile:linenumber` for C code and assembly function name for assembler code.

Callsites that do not provide callsite information are accumulated into an "unknown" entry.

failat=<*number of allocations*>
Causes memory allocation to fail (return NULL) after <*number of allocations*>. Setting <*number of allocations*> to 13 causes the 14th allocation to return NULL. De-allocations are not counted. Use this option to ensure that JVM code reliably handles allocation failures. This option is useful for checking allocation site behavior rather than setting a specific allocation limit.

ignoreUnknownBlocks
Ignores attempts to free memory that was not allocated using the **-Xcheck:memory** tool. Instead, the **-Xcheck:memory** statistics that are printed out at the end of a run indicates the number of “unknown” blocks that were freed.

mprotect=<*top*|*bottom*>
Locks pages of memory on supported platforms, causing the program to stop if padding before or after the allocated block is accessed for reads or writes. An extra page is locked on each side of the block returned to the user.

If you do not request an exact multiple of one page of memory, a region on one side of your memory is not locked. The **top** and **bottom** options control which side of the memory area is locked. **top** aligns your memory blocks to the top of the page (lower address), so buffer underruns result in an application failure. **bottom** aligns your memory blocks to the bottom of the page (higher address) so buffer overruns result in an application failure.

Standard padding scans detect buffer underruns when using **top** and buffer overruns when using **bottom**.

nofree
Keeps a list of blocks already used instead of freeing memory. This list, and the list of currently allocated blocks, is checked for memory corruption on every allocation and deallocation. Use this option to detect a dangling pointer (a pointer that is "dereferenced" after its target memory is freed). This option cannot be reliably used with long-running applications (such as WebSphere Application Server), because “freed” memory is never reused or released by the JVM.

noscan
Checks for blocks that are not freed. This option has little effect on performance, but memory corruption is not detected. This option is compatible only with **subAllocator**, **callsite**, and **callsitesmall**.

quick
Enables block padding only and is used to detect basic heap corruption. Every allocated block is padded with sentinel bytes, which are verified on every allocate and free. Block padding is faster than the default of checking every block, but is not as effective.

skipto=<*number of allocations*>
Causes the program to check only on allocations that occur after <*number of allocations*>. De-allocations are not counted. Use this option to speed up JVM startup when early allocations are not causing the memory problem. The JVM performs approximately 250+ allocations during startup.

subAllocator[=<*size in MB*>]
Allocates a dedicated and contiguous region of memory for all JVM allocations. This option helps to determine whether user JNI code or the JVM is responsible for memory corruption. Corruption in the JVM **subAllocator** heap suggests that the JVM is causing the problem; corruption in the user-allocated memory suggests that user code is corrupting memory. Typically, user and JVM allocated memory are interleaved.

zero
Newly allocated blocks are set to 0 instead of being filled with the `0xE7E7xxxxxxxxE7E7` pattern. Setting these blocks to 0 helps you to determine whether a callsite is expecting zeroed memory, in which case the allocation request is followed by `memset(pointer, 0, size)`.

Note: The **-Xcheck:memory** option cannot be used in the **-Xoptionsfile**.

-Xcheck:vm[:<*option*>]
Runs additional checks on the JVM. By default, no checks are made. For more information, run **-Xcheck:vm:**`help`.

-XcompilationThreads

You can change the number of threads used during JIT compilation with this option.

Purpose

This option allows you to specify the number of compilation threads used by the JIT compiler. The number of threads must be in the range 1 - 4, inclusive. Any other value prevents the JVM from starting successfully.

Setting the compilation threads to zero does not prevent the JIT from working. Instead, if you do not want the JIT to work, use the -Xint option.

When multiple compilation threads are used, the JIT might generate several diagnostic log files. A log file is generated for each compilation thread. The naming convention for the log file generated by the first compilation thread follows the same pattern as for IBM SDK, Java Technology Edition, Version 6:

```
<specified_filename>.<date>.<time>.<pid>
```

The first compilation thread has ID 0. Log files generated by the second and subsequent compilation threads append the ID of the corresponding compilation thread as a suffix to the log file name. The pattern for these log file names is as follows:

```
<specified_filename>.<date>.<time>.<pid>.<compThreadID>
```

For example, the second compilation thread has ID 1. The result is that the corresponding log file name has the form:

```
<specified_filename>.<date>.<time>.<pid>.1
```

Parameters

<number_of_threads>
Specifies the number of compilation threads. This number must be an integer value in the range 1 - 4, inclusive. Any other value prevents the JVM from starting successfully.

Use the following option to specify that the JIT compiler uses two compilation threads:

```
-XcompilationThreads2
```

-Xcompressedrefs and -Xnocompressedrefs (64-bit only)

Specify the **-Xcompressedrefs** option, on 64-bit operating systems, to use 32-bit values for references.

When using compressed references, the JVM stores all references to objects, classes, threads, and monitors as 32-bit values. Use the **-Xcompressedrefs** command-line option to enable compressed references in a 64-bit JVM. Use the **-Xnocompressedrefs** command-line option to disable compressed references. Only 64-bit JVMs recognize these options.

From this release, the **-Xcompressedrefs** option is the default setting for all operating systems other than z/OS, when the value of the **-Xmx** option is less than or equal to 25 GB. For z/OS operating systems, or values of **-Xmx** that are greater than 25 GB, compressed references are still disabled by default.

For more information about these options, see JVM command-line options in the diagnostic guide for Version 6.

-Xconcurrentlevel

Use the **-Xconcurrentlevel** option to modify memory allocation options.

Purpose

Specifies the allocation “tax” rate.

Parameters

-Xconcurrentlevel<*number*>

<*number*> indicates the ratio between the amount of heap allocated, and the amount of heap marked. The default ratio is 8.

To turn off concurrent marking, set <*number*> to the value 0.

Example

1. To turn off concurrent marking, use **-Xconcurrentlevel0**.

-Xjni

Sets JNI options.

-Xjni:<*suboptions*>

You can use the following suboption with the **-Xjni** option:

-Xjni:arrayCacheMax=[<*size in bytes*>|unlimited]

Sets the maximum size of the array cache. The default size is 128 KB.

-Xlog

Use the **-Xlog** option to modify the types of messages that the JVM writes to the system log. Changes do not affect messages written to the standard error stream (stderr).

Purpose

By default, all error and vital messages issued by the JVM are logged. You can change the default by using the **-Xlog** option.

Parameters

-Xlog[:*help*]|[:<*options*>]

Optional parameters are:

help

Details the options available.

error

Turns on logging for all error messages (default).

vital

Turns on logging for selected information messages JVMDUMP006I, JVMDUMP032I, and JVMDUMP033I, which provide valuable additional information about dumps produced by the JVM (default).

info

Turns on logging for all information messages.

warn

Turns on logging for all warning messages.

config

Turns on logging for all configuration messages.

all

Turns on logging for all messages.

none

Turns off logging for all messages.

The options **all none**, and **help** must be used on their own and cannot be combined. However, the other options can be grouped.

To obtain detailed information about logged messages, see the IBM SDK for Java Messages guide: http://www.ibm.com/support/knowledgecenter/SSYKE2_6.0.0/welcome/welcome_javasdk_version.html.

Examples

1. To include error, vital and warning messages use **-Xlog:error,vital,warn**.
2. To turn off message logging use **-Xlog:none**.

-Xlockword

The **-Xlockword** option enables performance improvements.

Purpose

This tuning option is available to test whether performance optimizations are negatively impacting an application. See "Application performance issues" on page 59

Parameters

The following parameters are available:

-Xlockword:mode=all
This option reverts to behavior that is closer to earlier versions.

-Xlockword:default
This option reestablishes the new behavior.

-Xlockword:nolockword=<*class_name*>
This option removes the lockword from object instances of the class <*class_name*>, reducing the space required for these objects. However, this action might have an adverse effect on synchronization for those objects. You should not use this option unless you are directed to by IBM service.

-Xlp

Use the **-Xlp** option to request the allocation of large pages.

Purpose

This option requests the JVM to allocate the Java object heap or the JIT code cache by using large pages.

Parameters

The following parameters are available:

-Xlp:codecache:pagesize=<size> (AIX, Linux, and Windows
-Xlp:codecache:pagesize=<size>,pageable (z/OS)

Requests the JVM to allocate the JIT code cache by using large page sizes. If the requested large page size is not available, the JVM starts, but the JIT code cache is allocated by using a platform-defined size. A warning is displayed when the requested page size is not available.

For service refresh 4, **-Xlp:codecache:pagesize=<size>** is supported on Linux on x86, Linux on z Systems, and Windows only.

To obtain the large page sizes available and the current setting, use the **-verbose:sizes** option. Note the current settings are the requested sizes and not the sizes obtained.

AIX: The code cache page size is controlled by the DATAPSIZE setting of the **LDR_CNTRL** environment variable. The page size cannot be controlled by the **-Xlp:codecache:pagesize=<size>** option. Specifying any other page size results in a warning that the page size is not available. The **-verbose:sizes** output reflects the current operating system setting. For more information about the **LDR_CNTRL** environment variable, see: **Working with the LDR_CNTRL environment variable**.

Linux PPC: The code cache page size cannot be controlled by the **-Xlp:codecache:pagesize=<size>** option. Specifying any other page size results in a warning that the page size is not available. The **-verbose:sizes** output reflects the current operating system setting.

z/OS: The **-Xlp:codecache:pagesize=<size>,pageable** option supports only a large page size of 1 M. The use of 1 M pageable large pages for the JIT code cache can improve the runtime performance of some Java applications. A page size of 4 K can also be used.

For more information, see "Configuring large page memory allocation" on page 37.

-Xlp:objectheap:pagesize=<*size*>,[strict],[warn] (AIX, Linux and Windows)
-Xlp:objectheap:pagesize=<*size*>,[strict],[warn],[non]pageable (z/OS)

Where:

- *<size>* is the large page size that you require for the Java object heap.
- `strict` causes an error message to be generated if large pages are requested but cannot be obtained. This suboption causes the JVM to end.
- `warn` causes a warning message to be generated if large pages are requested but cannot be obtained. This suboption allows the JVM to continue.

Note: If both suboptions are specified, `strict` overrides `warn`.

If the operating system does not have sufficient resources to satisfy the request, the page size you requested might not be available when the JVM starts up. By default, the JVM starts and the Java object heap is allocated by using a different platform-defined page size. Alternatively, you can use the `strict` or `warn` suboptions to customize behavior.

To obtain the large page sizes available and the current setting, use the **-verbose:sizes** option. Note the current settings are the requested sizes and not the sizes obtained. For object heap size information, check the **-verbose:gc** output.

z/OS: The `[non]pageable` argument defines the type of memory to allocate for the Java object heap.

Supported page sizes are 2G nonpageable, 1 M nonpageable, and 1 M pageable. A page size of 4 K can also be used.

All platforms: If you are running an earlier version that does not include the `strict` or `warn` suboptions, an error message is not generated when there are insufficient resources available. This limitation and a workaround for verifying which page size is used can be found in Known limitations.

For more information, see "Configuring large page memory allocation" on page 37.

-Xlp[<*size*>]

AIX: Requests the JVM to allocate the Java object heap (the heap from which Java objects are allocated) with large (16 MB) pages, if a size is not specified. If large pages are not available, the Java object heap is allocated with the next smaller page size that is supported by the system.

Linux: Requests the JVM to allocate the Java object heap by using large page sizes. If large pages are not available, the JVM does not start, displaying an error message. The JVM uses shmget() to allocate large pages for the heap. Large pages are supported by systems with Linux kernels v2.6 or higher. By default, large pages are not used.

If a <*size*> is specified, the JVM attempts to allocate the JIT code cache memory by using pages of that size. If unsuccessful, or if executable pages of that size are not supported, the JIT code cache memory is allocated by using the default or smallest available executable page size.

Note: Linux for System z® supports only a large page size of 1 M.

Windows: Requests the JVM to allocate the Java object heap with large pages. This command is available on Windows Server 2003 and later, and Windows Vista and later releases.

If a <*size*> is specified, the JVM attempts to allocate the JIT code cache memory by using pages of that size. If unsuccessful, or if executable pages of that size are not supported, the JIT code cache memory is allocated by using the default or smallest available executable page size.

z/OS: Requests the JVM to allocate the Java object heap by using large page sizes. If <*size*> is not specified, the 1M nonpageable size is used. If large pages are not supported by the hardware, or enabled in RACF, the JVM does not start and produces an error message.

Allocating large pages by using **-Xlp[<size>]** is only supported on the 64-bit SDK for z/OS, not the 31-bit JVM for z/OS.

If a <*size*> is specified, the JVM attempts to allocate the JIT code cache memory by using pages of that size. If unsuccessful, or if executable pages of that size are not supported, 1 M pageable is attempted. If 1 M pageable is not available, the JIT code cache memory is allocated using the default or smallest available executable page size.

On z/OS, **-Xlp**[<*size*>] supports only a large page size of 2G and 1 M (nonpageable). If the <*size*> parameter is not specified, 1 M (nonpageable) is used.

All platforms: To obtain the large page sizes available and the current setting, use the **-verbose:sizes** option. Note the current settings are the requested sizes and not the sizes obtained. For object heap size information, check the **-verbose:gc** output.

The JVM ends if there are insufficient operating system resources to satisfy the request. However, an error message is not issued. This limitation and a workaround for verifying which page size is used can be found in Known limitations.

For more information, see “Configuring large page memory allocation” on page 37.

-Xscdmx

Controls the memory required for storing the class debug data area used by shared classes.

Purpose

You can use the **-Xscdmx** option to control the size of the class debug area when creating a shared class cache. The **-Xscdmx** option works in a similar way to the **-Xscmx** option used to control the overall size of the shared class cache. The size of **-Xscdmx** must not exceed the size of **-Xscmx**. By default, the size of the class debug area is a percentage of the free bytes in a newly created or empty cache. The **-Xscdmx** option provides the ability to tune the cache region size.

If you use the **-Xscdmx** option, additional information is provided in "printStats utility" on page 106 and "Cache performance" on page 102.

Using**-Xnolinenumbers** does not create a class debug area. However, a class debug area is still created if you use the **-Xscdmx** option with the **-Xnolinenumbers** option on the command line.

Parameters

-Xscdmx<*x*>
: Sets the size of the shared class cache debug attribute area to **<x>**, which is expressed as an absolute value.

Example

An example use of the **-Xscdmx** option is as follows:

```
$java -Xscdmx1m -Xshareclasses:name=jim,reset -version
```

-Xscmaxjitdata

Sets the maximum shared classes cache space reserved for JIT data.

Purpose

You can use the **-Xscmaxjitdata** option to set the maximum cache space reserved for JIT shared classes. The **-Xscmaxjitdata** option works in a similar way to the **-Xscmx** option used to control the overall size of the shared class cache.

If you use the **-Xscmaxjitdata** option, additional information is in the "printStats utility" on page 106.

Parameters

-Xscmaxjitdata<*x*>
: Optionally applies a maximum number of bytes in the class cache that can be used for JIT data. This option is useful if you want a certain amount of cache space guaranteed for non-JIT data. If this option is not specified, the maximum limit for JIT data is the amount of free space in the cache. The value of this option must not be smaller than the value of **-Xscminjitdata**, and must not be larger than the value of **-Xscmx**.

Example

An example use of the **-Xscmaxjitdata** option is as follows:

```
$java -Xscmaxjitdata1m -Xshareclasses:name=jim,reset -version
```

-Xscminjitdata

Sets the minimum shared classes cache space reserved for JIT data.

Purpose

You can use the **-Xscminjitdata** option to set the minimum cache space reserved for JIT shared classes. The **-Xscminjitdata** option works in a similar way to the **-Xscmx** option used to control the overall size of the shared class cache.

If you use the **-Xscminjitdata** option, additional information is in the "printStats utility" on page 106.

Parameters

-Xscminjitdata<*x*>

Optionally applies a minimum number of bytes in the class cache to reserve for JIT data. If this option is not specified, no space is reserved for JIT data, although JIT data is still written to the cache until the cache is full or the **-Xscmaxjitdata** limit is reached. The value of this option must not exceed the value of **-Xscmx** or **-Xscmaxjitdata**. The value of **-Xscminjitdata** must always be considerably less than the total cache size, because JIT data can be created only for cached classes. If the value of **-Xscminjitdata** equals the value of **-Xscmx**, no class data or JIT data can be stored.

Example

An example use of the **-Xscminjitdata** option is as follows:

```
$java -Xscminjitdata1m -Xshareclasses:name=jim,reset -version
```

-Xsignal:userConditionHandler=percolate (31-bit z/OS only)

This option is used to control 31-bit z/OS JVM Language Environment condition handling. The behavior is similar to that of the -XCEEHDLR option, but differs in the severity level of the affected Language Environment conditions.

As with the **-XCEEHDLR** option, the JVM registers user condition handlers to handle the z/OS exceptions that would otherwise be handled by the JVM POSIX signal handlers for the SIGBUS, SIGFPE, SIGILL, SIGSEGV, and SIGTRAP signals. The JVM does not install POSIX signal handlers for these signals. This option differs from the **-XCEEHDLR** option in that the JVM percolates **all** Language Environment conditions that were not triggered and expected by the JVM during normal running, including conditions that are severity 2 or greater. The JVM generates its own diagnostic information before percolating severity 2 or greater conditions.

Notes:

- The JVM is in an undefined state after percolating a severity 2 or greater condition. Applications cannot resume running then call back into, or return to, the JVM.
- This option is not compatible with the following options:
  - **-XCEEHDLR**
  - **-Xsignal:posixSignalHandler=cooperativeShutdown**

Related reference:

"-XCEEHDLR (31-bit z/OS only)" on page 139
The -XCEEHDLR option is used to control 31-bit z/OS JVM Language Environment condition handling.

-Xthr

The **-Xthr** option enables performance improvements.

Purpose

This tuning option is available to test whether performance optimizations are negatively impacting an application. See "Application performance issues" on page 59

Parameters

The following parameters are available:

-Xthr:<AdaptSpin|noAdaptSpin>
These options are used to turn on or off a specific optimization.

-Xthr:<cfsYield|noCfsYield> (Linux only)
The default value, **cfsYield**, enables threading optimizations for applications running on Linux with the Completely Fair Scheduler (CFS) in the default mode (`sched_compat_yield=0`). The **noCfsYield** value disables these threading optimizations. You might want to use the **noCfsYield** value if your application uses the Thread.yield() method extensively, because otherwise you might see a performance decrease in cases where yielding is not beneficial.

-Xthr:<secondarySpinForObjectMonitors|noSecondarySpinForObjectMonitors>
These options are used to turn on or off a specific optimization.

-Xtune

The **-Xtune** option enables performance improvements.

Purpose

This tuning option is available to optimize JVM performance.

Parameters

The following parameters are available:

-Xtune:elastic
This option turns on JVM function that accommodates changes in the machine configuration dynamically at run time. Such changes might include the number of processors, or the amount of installed RAM.

Note: From service refresh 7, this option has no effect.

-Xzero

Reduces the memory footprint of the JVM when running multiple JVM invocations concurrently.

Purpose

You can use the **-Xzero** option to reduce the amount of memory used by your runtime environment when you are running multiple JVM invocations at the same time. This option might not be appropriate for all types of applications because it changes the implementation of `java.util.ZipFile`, which might cause extra memory usage.

Parameters

-Xzero[:<*option*>**]**

Optional parameters are:

j9zip
Enables the **j9zip** sub-option.

noj9zip
Enables the **noj9zip** sub-option.

sharezip
Enables the **sharezip** sub-option.

nosharezip
Enables the **nosharezip** sub-option.

sharebootzip
Enables the **sharebootzip** sub-option (default).

nosharebootzip
Enables the **nosharebootzip** sub-option.

none
disables all sub-options.

describe
Prints the sub-options in effect.

Because future versions might include more default options, **-Xzero** options are used to specify the sub-options that you want to disable. By default, **-Xzero** enables **j9zip** and **sharezip**. A combination of **j9zip** and **sharezip** enables all jar files to have shared caches:

- **j9zip** - uses a new `java.util.ZipFile` implementation. This sub-option is not a requirement for **sharezip**; however, if **j9zip** is not enabled, only the bootstrap jars have shared caches.
- **sharezip** - puts the j9zip cache into shared memory. The j9zip cache is a map of .zip entry names to file positions, used to quickly find entries in the .zip file. You must enable **-Xshareclasses** to avoid a warning message. When using the **sharezip** sub-option, note that every opened .zip file and .jar file stores the j9zip cache in shared memory. You might fill the shared memory when opening multiple new .zip files and .jar files. The affected API is `java.util.zip.ZipFile` (superclass of `java.util.jar.JarFile`). The .zip and .jar files do not have to be on a class path.
- **sharebootzip** - enabled by default on all platforms. Puts the .zip entry caches for bootstrap jar files into the shared cache. A .zip entry cache is a map of .zip entry names to file positions, used to quickly find entries in the .zip file.

The system property com.ibm.zero.version is defined, and has a current value of 2. Although **-Xzero** is accepted on all platforms, support for the sub-options varies by platform:

- **-Xzero** with the **sharebootzip** and **nosharebootzip** sub-options are accepted on all platforms.
- **-Xzero** with all other sub-options are available only on Windows x86-32 and Linux x86-32 platforms.

-XX command-line options

JVM command-line options that are specified with -XX are not recommended for casual use.

These options are subject to change without notice.

To see a complete list of JVM -XX command-line options, see the Java 6 Diagnostics Guide, ../../../com.ibm.java.doc.diagnostics.60/diag/appendixes/cmdline/commands_jvm_xx.html.

-XX:[+|-]handleSIGXFSZ:

This option is available only on the Linux platform and affects the handling of the operating system signal `SIGXFSZ`. This signal is generated when a process attempts to write to a file that causes the maximum file size `ulimit` to be exceeded. If the signal is not handled by the JVM, the operating system ends the process with a core dump.

+XX:+handleSIGXFSZ
When this option is set, the JVM handles the signal `SIGXFSZ` and continues, without ending. When a file is written from a Java API class that exceeds the maximum file size `ulimit`, an exception is raised. Log files that are created by the JVM are silently truncated when they reach the maximum file size `ulimit`.

-XX:-handleSIGXFSZ
When this option is set, the JVM does not handle the signal `SIGXFSZ`. If the maximum file size `ulimit` for any file is reached, the operating system ends the process with a core dump. This option is the default.

-XX:[+|-]LazySymbolResolution (Linux only):

The **-XX:+LazySymbolResolution** option forces the JVM to delay symbol resolution for each function in a user native library, until the function is called. This option is the default setting.

The **-XX:-LazySymbolResolution** option forces the JVM to immediately resolve symbols for all functions in a user native library when the library is loaded.

These options apply only to functions; variable symbols are always resolved immediately when loaded. If you attempt to use these options on an operating system other than Linux, the options are accepted, but ignored.

-XXnosuballoc32bitmem (z/OS):

When using compressed references on a 64-bit JVM, use the **-XXnosuballoc32bitmem** option to force the JVM to use 31-bit memory allocation functions provided by the operating system.

Purpose

This option is provided as a workaround for customers who need to use fewer pages of 31-bit virtual storage per JVM invocation. Using this option might result in a small increase in the number of frames of central storage used by the JVM. However, the option frees 31-bit pages for use by native code or other applications in the same address space.

If you do not use this option, the JVM uses an allocation strategy for 31-bit memory that reserves a region of 31-bit virtual memory. However, because the **-XXnosuballoc32bitmem** option forces the JVM to use the z/OS allocation strategy, virtual memory is not reserved by the JVM.

-XXsetHWPrefetch:[none|os-default] (AIX only):

The **-XXsetHWPrefetch:none** option disables hardware prefetch. Hardware prefetch can improve the performance of applications by prefetching memory, however because of the workload characteristics of many Java applications, prefetching often has an adverse effect on performance.

You can disable hardware prefetch on AIX by issuing the command `dscrctl -n -s 1`. However, this command disables hardware prefetch for all processes, and for all future processes, which might not be desirable in a mixed workload environment. Instead, you can use the **-XXsetHWPrefetch:none** option to disable hardware prefetch for individual JVMs. The default behavior is to use the hardware prefetch setting of the operating system.

From service refresh 5, you can revert to the default behavior by using the **-XXsetHWPrefetch:os-default** option. Use this option to override a **-XXsetHWPrefetch:none** setting that you previously specified on the command line.

-XX:ShareClassesEnableBCI:

This option is equivalent to **-Xshareclasses:enableBCI**.

Purpose

-XX:ShareClassesEnableBCI can be specified for any version of the IBM J9 virtual machine, but is ignored by JVMs that are earlier than the IBM J9 2.6 virtual machine. If BCI support is enabled with this option, you can turn off BCI support with **-Xshareclasses:disableBCI**.

For more information about **-Xshareclasses:enableBCI** and **-Xshareclasses:disableBCI**, see "-Xshareclasses" on page 155.

-XX:[+|-]VMLockClassLoader:

This option affects synchronization on class loaders that are not parallel-capable class loaders, during class loading.

-XX:[+|-]VMLockClassLoader

The option, **-XX:+VMLockClassLoader**, causes the JVM to force synchronization on a class loader that is not a parallel capable class loader during class loading. This action occurs even if the loadClass() method for that classloader is not synchronized. For information about parallel capable class loaders, see java.lang.ClassLoader.registerAsParallelCapable() in Java 7. Note that this option might cause a deadlock if classloaders use non-hierarchical delegation. For example, setting the system property **osgi.classloader.lock=classname** with Equinox is known to cause a deadlock.

When specifying the **-XX:-VMLockClassLoader** option, the JVM does not force synchronization on a class loader during class loading. The class loader still conforms to class library synchronization, such as a synchronized loadClass() method. This is the default option, which might change in future releases.

Class data sharing command-line options

These command-line options can be used to configure shared classes.

This reference section provides a list of command-line options that are new, or changed, when IBM SDK, Java Technology Edition, Version 6 uses an IBM J9 2.6 virtual machine. To see a complete list of class data sharing command-line options, see the Java 6 Diagnostics Guide, ../../../com.ibm.java.doc.diagnostics.60/diag/appendixes/cmdline/commands_jvm.html

-Xscdmx

Controls the memory required for storing the class debug data area used by shared classes.

Purpose

You can use the **-Xscdmx** option to control the size of the class debug area when creating a shared class cache. The **-Xscdmx** option works in a similar way to the **-Xscmx** option used to control the overall size of the shared class cache. The size of **-Xscdmx** must not exceed the size of **-Xscmx**. By default, the size of the class debug area is a percentage of the free bytes in a newly created or empty cache. The **-Xscdmx** option provides the ability to tune the cache region size.

If you use the **-Xscdmx** option, additional information is provided in "printStats utility" on page 106 and "Cache performance" on page 102.

Using**-Xnolinenumbers** does not create a class debug area. However, a class debug area is still created if you use the **-Xscdmx** option with the **-Xnolinenumbers** option on the command line.

Parameters

-Xscdmx<*x*>
: Sets the size of the shared class cache debug attribute area to **<x>**, which is expressed as an absolute value.

Example

An example use of the **-Xscdmx** option is as follows:

```
$java -Xscdmx1m -Xshareclasses:name=jim,reset -version
```

-Xshareclasses

Controls class data sharing between JVMs. There are a number of changes from IBM SDK, Java Technology Edition, Version 6.

Purpose

You can use the **-Xshareclasses** option to control class data sharing. For a full list of options, see the IBM SDK, Java Technology Edition, Version 6 topic -Xshareclasses option.

Parameters

-Xshareclasses:<*suboption*>[,<*suboption*>...]
: where <*suboption*> includes the following changes from IBM SDK, Java Technology Edition, Version 6:

  cacheDirPerm=<*permission*>
  : Available only on AIX, UNIX and z/OS operating systems. Sets UNIX-style permissions when creating a cache directory. <*permission*> must be an octal

number in the ranges 0700 - 0777 or 1700 - 1777. If <*permission*> is not valid, the JVM terminates with an appropriate error message.

The permissions specified by this suboption are used only when creating a new cache directory. If the cache directory already exists, this suboption is ignored and the cache directory permissions are not changed.

If you set this suboption to 0000, the default directory permissions are used. If you set this suboption to 1000, the machine default directory permissions are used, but the sticky bit is enabled.If the cache directory is the platform default directory, `/tmp/javasharedresources`, this suboption is ignored and the cache directory permissions are set to 777. If you do not set this suboption, the cache directory permissions are set to 777, for compatibility with earlier Java versions.

disableBCI

Turns off BCI support. This option can be used to override "-XX:ShareClassesEnableBCI" on page 154.

enableBCI

Allows a JVMTI ClassFileLoadHook event to be triggered every time, for classes loaded from the cache. This mode also prevents caching of classes modified by JVMTI agents. For more information about this option, see "Using the JVMTI ClassFileLoadHook with cached classes" on page 104. This option is incompatible with the **cacheRetransformed** option. Using the two options together causes the JVM to end with an error message, unless **-Xshareclasses:nonfatal** is specified. In this case, the JVM continues without using shared classes.

This mode stores more data into the cache, and creates a Raw Class Data area by default. See the **rcdSize=** suboption. When using this suboption, the cache size might need to be increased with **-Xscmx<size>**.

A cache created without the **enableBCI** suboption cannot be reused with the **enableBCI** suboption. Attempting to do so causes the JVM to end with an error message, unless **-Xshareclasses:nonfatal** is specified. In this case, the JVM continues without using shared classes. A cache created with the **enableBCI** suboption can be reused without specifying this suboption. In this case, the JVM detects that the cache was created with the **enableBCI** suboption and uses the cache in this mode.

findAotMethods=help|{<method_specification>[,<method_specification>]}
(Utility option)
Print the AOT methods in the shared cache that match the method specifications. Methods that are already invalidated are indicated in the output. Use this suboption to check which AOT methods in the shared class cache would be invalidated by using the same method specifications with the **invalidateAotMethods** suboption. To learn more about the syntax to use when you are specifying more than one method specification, see "Method specification syntax" on page 159.

invalidateAotMethods=help|{<method_specification>[,<method_specification>]}
(Utility option)
Modify the existing shared cache to invalidate the AOT methods matching the method specifications. Use this suboption to invalidate AOT methods that cause a failure in the application, without having to destroy the shared cache. Invalidated AOT methods remain in the shared cache, but are then excluded from being loaded. JVMs that have not processed the methods, or new JVMs that use the cache are not affected by the invalidated methods.

The AOT methods are invalidated for the lifetime of the cache, but do not prevent the AOT methods from being compiled again if a new shared cache is created. To prevent AOT method compilation into a new shared cache, use the **-Xaot:exclude=** option. For more information, see -Xaot. To identify AOT problems, see Diagnosing JIT or AOT problems. To revalidate an AOT method, see the **revalidateAotMethods** suboption. Use the **findAotMethod** suboption to determine which AOT methods match the method specifications. To learn more about the syntax to use when you are specifying more than one method specification, see "Method specification syntax" on page 159.

mprotect=[default | all | partialpagesonstartup | onfind | nopartialpages | none]

Where:

- **default**: By default, the memory pages that contain the cache are always protected, unless a specific page is being updated. This protection helps prevent accidental or deliberate corruption to the cache. The cache header is not protected by default because this protection has a performance cost.

  After the startup phase, the Java virtual machine (VM) protects partially filled pages whenever new data is added to the shared class cache in the following sequence:

  – The VM changes the memory protection of any partially filled pages to read/write.
  – The VM adds the data to the cache.
  – The VM changes the memory protection of any partially filled pages to read only.

  The protection of partially filled pages is introduced in service refresh 8 fix pack 20 on Linux and Windows platforms.
- **all**: This option ensures that all the cache pages are protected, including the header.
- **partialpagesonstartup**: This option causes the JVM to protect partially filled pages during startup as well as after the startup phase
- **onfind**: When this option is specified, the JVM protects partially filled pages when it reads new data in the cache that is added by another JVM.
- **nopartialpages**: Use this option to turn off the protection of partially filled pages
- **none**: Specifying this option disables the page protection.

Note: Specifying **all** has a negative impact on performance. You should specify **all** only for problem diagnosis and not for production. Specifying **partialpagesonstartup** or **onfind** options can also have a negative impact on performance when the cache is being populated. There is no further impact when the cache is full or no longer being modified.

none

Can be added to the end of a command line to disable class data sharing. This suboption overrides class sharing arguments found earlier on the command line. This suboption disables the shared class utility APIs. To disable class data sharing without disabling utility APIs, use the **utilities** suboption. For more information about the shared class utility APIs, see "Utility APIs" on page 105.

persistent
Uses a persistent cache. The cache is created on disk, which persists beyond operating system restarts. Non-persistent and persistent caches can have the same name.

AIX: In this release, the **persistent** suboption is the default setting, and you no longer have to use the persistent suboption when running utilities such as **destroy** on a persistent cache.

rcdSize=*nnn*

Controls the size of the Raw Class Data Area. The number of bytes passed to **rcdSize** must always be less than the total cache size. This value is always rounded down to the nearest multiple of the system page size. For example, these variations specify a Raw Class Data Area with a size of 1 MB:

```
-Xshareclasses:enableBCI,rcdSize=1048576
-Xshareclasses:enableBCI,rcdSize=1024k
-Xshareclasses:enableBCI,rcdSize=1m
```

If **rcdSize** is not used, and **enableBCI** is used, the JVM chooses a default Raw Class Data Area size.

If **rcdSize** is used, memory is reserved in the cache regardless of whether **enableBCI** is used.

If neither **rcdSize** or **enableBCI** is used, nothing is reserved in the cache for the Raw Class Data Area.

revalidateAotMethods=help|{<method_specification>[,<method_specification>]} (Utility option)
Modify the shared cache to revalidate the AOT methods that match the method specifications. Use this suboption to revalidate AOT methods that were invalidated by using the **invalidateAotMethods** suboption. Revalidated AOT methods are then eligible for loading into a JVM, but do not affect JVMs where the methods have already been processed. To learn more about the syntax to use when you are specifying more than one method specification, see "Method specification syntax" on page 159.

safemode
This option is no longer recognized. If you want to turn off class data sharing, use the **none** option instead.

utilities
Can be added to the end of a command line to disable class data sharing. This suboption overrides class sharing arguments found earlier on the command line. This suboption is like **none**, but does not disable the shared class utility APIs. For more information about the shared class utility APIs, see "Utility APIs" on page 105.

printStats[=<*data_types*>] (Utility option)
Displays summary information for the cache specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. The most useful information displayed is how full the cache is and how many classes it contains. Stale classes are classes that have been updated on the file system and which the cache has therefore marked "stale". Stale classes are not purged from the cache and can be reused.

Specify one or more data types, separated by a plus symbol (+), to additionally see more detailed information about that type of cache content. Data types include AOT data, class paths and ROMMethods.See "printStats utility" on page 106 for more information.

nojitdata
Disables caching of JIT data. JIT data already in the shared data cache can be loaded.

Example

An example use of the **-Xshareclasses** option is as follows:

```
$java -Xshareclasses:name=jim,nojitdata
```

Method specification syntax

The following examples show how to specify more than one method specification when you are using the **findAotMethods=**, **invalidateAotMethods=**, or **revalidateAotMethods=** suboptions.

Braces, {}, are required around the method specification if you specify more than one method specification. If the specification contains a comma, <method_specification> is defined as the following string:

```
[!]{*|[*]<packagename/classname>[*]}[.{*|[*]<methodname>[*]}][({*|[*]<parameters>[*]})]
```

Parameters are optional, but if specified, the following native signature formats must be used:

- B for byte
- C for char
- D for double
- F for float
- I for int
- J for long
- S for short
- Z for Boolean
- L<class name>; for objects
- [before the signature means array

If parameters must be specified to distinguish the method, use **findAotMethods=** with the string (*) to list all the parameter variations. Copy the signature for the method that you want from the output. For example, the signature for the parameters (byte[] bytes, int offset, int length, Charset charset) is ([BIILjava/nio/charset/Charset;).

Here are some examples:

- * - matches all AOT methods.
- java/lang/Object - matches all AOT methods in the java.lang.Object class.
- java/util/* - matches all AOT classes and methods in the java.util package.
- java/util/HashMap.putVal - matches all putVal methods in the java.util.HashMap class.
- java/util/HashMap.hash(Ljava/lang/Object;) - matches the private java.util.HashMap.hash(java.lang.Object method.
- *.equals - matches all equals methods in all classes.
- {java/lang/Object,!java/lang/Object.clone} - matches all methods in java.lang.Object except clone.

- {java/util/*.*(),java/lang/Object.*(*)} - matches all classes or methods with no input parameter in the java.util package, and all methods in java.lang.Object.
- {java/util/*.*(),!java/util/*.*()} - matches nothing.

-Xscmaxjitdata

Sets the maximum shared classes cache space reserved for JIT data.

Purpose

You can use the **-Xscmaxjitdata** option to set the maximum cache space reserved for JIT shared classes. The **-Xscmaxjitdata** option works in a similar way to the **-Xscmx** option used to control the overall size of the shared class cache.

If you use the **-Xscmaxjitdata** option, additional information is in the "printStats utility" on page 106.

Parameters

-Xscmaxjitdata<*x*>

Optionally applies a maximum number of bytes in the class cache that can be used for JIT data. This option is useful if you want a certain amount of cache space guaranteed for non-JIT data. If this option is not specified, the maximum limit for JIT data is the amount of free space in the cache. The value of this option must not be smaller than the value of **-Xscminjitdata**, and must not be larger than the value of **-Xscmx**.

Example

An example use of the **-Xscmaxjitdata** option is as follows:

```
$java -Xscmaxjitdata1m -Xshareclasses:name=jim,reset -version
```

-Xscminjitdata

Sets the minimum shared classes cache space reserved for JIT data.

Purpose

You can use the **-Xscminjitdata** option to set the minimum cache space reserved for JIT shared classes. The **-Xscminjitdata** option works in a similar way to the **-Xscmx** option used to control the overall size of the shared class cache.

If you use the **-Xscminjitdata** option, additional information is in the "printStats utility" on page 106.

Parameters

-Xscminjitdata<*x*>

Optionally applies a minimum number of bytes in the class cache to reserve for JIT data. If this option is not specified, no space is reserved for JIT data, although JIT data is still written to the cache until the cache is full or the **-Xscmaxjitdata** limit is reached. The value of this option must not exceed the value of **-Xscmx** or **-Xscmaxjitdata**. The value of **-Xscminjitdata** must always be considerably less than the total cache size, because JIT data can be created only for cached classes. If the value of **-Xscminjitdata** equals the value of **-Xscmx**, no class data or JIT data can be stored.

Example

An example use of the **-Xscminjitdata** option is as follows:

```
$java -Xscminjitdata1m -Xshareclasses:name=jim,reset -version
```

-Xzero

Reduces the memory footprint of the JVM when running multiple JVM invocations concurrently.

Purpose

You can use the **-Xzero** option to reduce the amount of memory used by your runtime environment when you are running multiple JVM invocations at the same time. This option might not be appropriate for all types of applications because it changes the implementation of `java.util.ZipFile`, which might cause extra memory usage.

Parameters

-Xzero[:<*option*>**]**

Optional parameters are:

j9zip
Enables the **j9zip** sub-option.

noj9zip
Enables the **noj9zip** sub-option.

sharezip
Enables the **sharezip** sub-option.

nosharezip
Enables the **nosharezip** sub-option.

sharebootzip
Enables the **sharebootzip** sub-option (default).

nosharebootzip
Enables the **nosharebootzip** sub-option.

none
disables all sub-options.

describe
Prints the sub-options in effect.

Because future versions might include more default options, **-Xzero** options are used to specify the sub-options that you want to disable. By default, **-Xzero** enables **j9zip** and **sharezip**. A combination of **j9zip** and **sharezip** enables all jar files to have shared caches:

- **j9zip** - uses a new `java.util.ZipFile` implementation. This sub-option is not a requirement for **sharezip**; however, if **j9zip** is not enabled, only the bootstrap jars have shared caches.
- **sharezip** - puts the j9zip cache into shared memory. The j9zip cache is a map of .zip entry names to file positions, used to quickly find entries in the .zip file. You must enable **-Xshareclasses** to avoid a warning message. When using the **sharezip** sub-option, note that every opened .zip file and .jar file stores the j9zip cache in shared memory. You might fill the shared memory when opening multiple new .zip files and .jar files. The affected

API is `java.util.zip.ZipFile` (superclass of `java.util.jar.JarFile`). The .zip and .jar files do not have to be on a class path.

- **sharebootzip** - enabled by default on all platforms. Puts the .zip entry caches for bootstrap jar files into the shared cache. A .zip entry cache is a map of .zip entry names to file positions, used to quickly find entries in the .zip file.

The system property com.ibm.zero.version is defined, and has a current value of 2. Although **-Xzero** is accepted on all platforms, support for the sub-options varies by platform:

- **-Xzero** with the **sharebootzip** and **nosharebootzip** sub-options are accepted on all platforms.
- **-Xzero** with all other sub-options are available only on Windows x86-32 and Linux x86-32 platforms.

-XX command-line options

JVM command-line options that are specified with -XX are not recommended for casual use.

These options are subject to change without notice.

To see a complete list of JVM -XX command-line options, see the Java 6 Diagnostics Guide, ../../../com.ibm.java.doc.diagnostics.60/diag/appendixes/cmdline/commands_jvm_xx.html.

-XX:ShareClassesEnableBCI:

This option is equivalent to **-Xshareclasses:enableBCI**.

Purpose

-XX:ShareClassesEnableBCI can be specified for any version of the IBM J9 virtual machine, but is ignored by JVMs that are earlier than the IBM J9 2.6 virtual machine. If BCI support is enabled with this option, you can turn off BCI support with **-Xshareclasses:disableBCI**.

For more information about **-Xshareclasses:enableBCI** and **-Xshareclasses:disableBCI**, see "-Xshareclasses" on page 155.

JIT and AOT command-line options

There are a number of command-line options used by the JVM Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilers.

This reference section provides a list of command-line options that are new, or changed, when IBM SDK, Java Technology Edition, Version 6 is used with an IBM J9 2.6 virtual machine. To see a complete list of JIT and AOT command-line options, see the Java 6 Diagnostics Guide, ../../../com.ibm.java.doc.diagnostics.60/diag/appendixes/cmdline/commands_jit.html.

-Xcodecachetotal

Use this option to set the maximum size limit for the JIT code cache. This option also affects the size of the JIT data cache.

-Xcodecachetotal<*size*>

For more information about this option, see -Xcodecachetotal in the diagnostic guide for Version 6.

When IBM SDK, Java Technology Edition, Version 6 Version 6 uses an IBM J9 2.6 virtual machine, this option also proportionally increases the maximum size limit for the JIT data cache, which holds metadata about compiled methods, to support the additional JIT compilations.

-XcompilationThreads

You can change the number of threads used during JIT compilation with this option.

Purpose

This option allows you to specify the number of compilation threads used by the JIT compiler. The number of threads must be in the range 1 - 4, inclusive. Any other value prevents the JVM from starting successfully.

Setting the compilation threads to zero does not prevent the JIT from working. Instead, if you do not want the JIT to work, use the -Xint option.

When multiple compilation threads are used, the JIT might generate several diagnostic log files. A log file is generated for each compilation thread. The naming convention for the log file generated by the first compilation thread follows the same pattern as for IBM SDK, Java Technology Edition, Version 6:

```
<specified_filename>.<date>.<time>.<pid>
```

The first compilation thread has ID 0. Log files generated by the second and subsequent compilation threads append the ID of the corresponding compilation thread as a suffix to the log file name. The pattern for these log file names is as follows:

```
<specified_filename>.<date>.<time>.<pid>.<compThreadID>
```

For example, the second compilation thread has ID 1. The result is that the corresponding log file name has the form:

```
<specified_filename>.<date>.<time>.<pid>.1
```

Parameters

<number_of_threads>
Specifies the number of compilation threads. This number must be an integer value in the range 1 - 4, inclusive. Any other value prevents the JVM from starting successfully.

Use the following option to specify that the JIT compiler uses two compilation threads:

```
-XcompilationThreads2
```

-Xquickstart

Use the **-Xquickstart** option to enable optimizations that are intended to improve performance.

Purpose

This option causes the JIT compiler to run with a subset of optimizations. The effect is faster compilation times that improve startup time. However, longer running applications might run slower. When the AOT compiler is active (both shared classes and AOT compilation enabled), **-Xquickstart** causes all methods to be AOT compiled. The AOT compilation improves the startup time of subsequent

runs, but might reduce performance for longer running applications. **-Xquickstart** can degrade performance if it is used with long-running applications that contain hot methods. The implementation of **-Xquickstart** is subject to change in future releases. By default, **-Xquickstart** is not enabled.

Another way to specify a behavior identical to **-Xquickstart** is to use the **-client** option. These two options can be used interchangeably on the command line.

Garbage collection command-line options

There are a number of command-line options used by JVM garbage collection operations. The options can apply to multiple garbage collection policies.

This reference section provides a list of command-line options that are new, or changed, when IBM SDK, Java Technology Edition, Version 6 is used with an IBM J9 2.6 virtual machine. To see a complete list of garbage collection command-line options, see the Java 6 Diagnostics Guide, ../../../com.ibm.java.doc.diagnostics.60/diag/appendixes/cmdline/commands_gc.html.

To see the new or changed policy options, see "Garbage collection policy options" on page 47.

-Xgc

Use the **-Xgc** option to tune garbage collection.

Purpose

The **-Xgc** option can be used with a number of parameters to fine-tune garbage collection. For a full list of options, see -Xgc option.

Parameters

-Xgc:minContractPercent=<n>

The minimum percentage of the heap that can be contracted at any given time.

-Xgc:maxContractPercent=<n>

The maximum percentage of the heap that can be contracted at any given time. For example, **-Xgc:maxContractPercent=20** causes the heap to contract by as much as 20%.

scvNoAdaptiveTenure

This option turns off the adaptive tenure age in the generational concurrent GC policy. The initial age that is set is maintained throughout the run time of the Java virtual machine. See **scvTenureAge**.

scvTenureAge=<*n*>

This option sets the initial scavenger tenure age in the generational concurrent GC policy. The range is 1 - 14 and the default value is 10. For more information about tenure age, see Tenure age

-Xgc:verboseFormat

Accepted values are:

- *default*: The default verbose garbage collection format available in the IBM J9 2.6 virtual machine. See "Verbose garbage collection logging" on page 111.
- *deprecated*: The verbose garbage collection format available in earlier releases of the J9 VM. For more information, see Verbose garbage collection logging.

-Xgcpolicy

Use the **-Xgcpolicy** option to specify the garbage collection policy you want to use.

Purpose

The Balanced garbage collection policy is new. You can specify this policy by using **-Xgcpolicy:balanced**. For information about the policies available, see "Garbage collection policy options" on page 47.

Parameters

-Xgcpolicy:balanced

Specifies the balanced garbage collection policy. For information about the garbage collection command-line options that can be used with the balanced policy, see "Balanced Garbage Collection policy options" on page 169.

-Xgcthreads

Use the **-Xgcthreads** option to set the number of threads that the Garbage Collector uses for parallel operations.

Purpose

Sets the number of threads that the Garbage Collector uses for parallel operations. This total number of GC threads is composed of one application thread with the remainder being dedicated GC threads. By default, the number is set to the number of physical CPUs present, up to a maximum of 64. To set it to a different number (for example 4), use **-Xgcthreads4**. The minimum valid value is 1, which disables parallel operations, at the cost of performance. No advantage is gained if you increase the number of threads beyond the default setting; you are recommended not to do so.

On systems running multiple JVMs or in LPAR environments where multiple JVMs can share the same physical CPUs, you might want to restrict the number of GC threads used by each JVM. The restriction helps prevent the total number of parallel operation GC threads for all JVMs exceeding the number of physical CPUs present, when multiple JVMs perform garbage collection at the same time.

Parameters

-Xgcthreads<*number*>

<*number*> is the number of threads to use for parallel operations.

-Xmcrs

Sets an initial size for an area in memory that is reserved for compressed references within the lowest 4 GB memory area.

Native memory OutOfMemoryError exceptions might occur when using compressed references if the lowest 4 GB of address space becomes full, particularly when loading classes, starting threads, or using monitors. This option secures space for any native classes, monitors, and threads that are used by compressed references.

-Xmcrs<*mem_size*>

Where <*mem_size*> is the initial size. You can use the **-verbose:sizes** option to find out the value that is being used by the VM. If you are not using compressed references and this option is set, the option is ignored and the output of **-verbose:sizes** shows `-Xmcrs0`.

The following option sets an initial size of 200 MB for the memory area:

```
-Xmcrs200M
```

-Xmn

The **-Xmn** option is equivalent to setting both the **-Xmns** and **-Xmnx** options when using **-Xgcpolicy:gencon** or **-Xgcpolicy:balanced**.

Purpose

When using the **-Xmn** option with the **-Xgcpolicy:gencon** or **-Xgcpolicy:balanced**, there are some important considerations:

- If you set either **-Xmns** or **-Xmnx**, you cannot set **-Xmn**. The JVM does not start and returns an error.
- When using **-Xgcpolicy:gencon** with the scavenger disabled, the **-Xmn** option is ignored.

Parameters

-Xmn<*size*>

where <*size*> is an absolute value.

-Xmns

The **-Xmns** option sets an initial size for the new area, or eden space, depending on the garbage collection policy specified.

Purpose

When using **-Xgcpolicy:gencon**, **-Xmns** sets the initial size of the new area. By default, this option is set to 25% of the value of the **-Xms** option. If the scavenger is disabled, the **-Xmns** option is ignored.

When using **-Xgcpolicy:balanced**, **-Xmns** sets the initial size of the eden space. By default, this option uses the smaller of these values:

- The value specified for the **-Xms** option.
- 25% of the value specified for the **-Xmx** option.

For both the **-Xgcpolicy:gencon** and **-Xgcpolicy:balanced** policies, the JVM returns an error if you try to use **-Xmns** with **-Xmn**.

To find the value of **-Xmns** that the JVM is using, specify the **-verbose:sizes** option on the command line.

Parameters

-Xmns<*size*>

where <*size*> is an absolute value.

-Xmnx

The **-Xmnx** option sets a maximum size for the new area, or eden space, depending on the garbage collection policy specified.

Purpose

When using **-Xgcpolicy:gencon**, **-Xmnx** sets the maximum size of the new area. By default, this option is set to 25% of the value of the **-Xmx** option. If the scavenger is disabled, the **-Xmnx** option is ignored.

When using **-Xgcpolicy:balanced**, **-Xmnx** sets the maximum size of the eden space. By default, this option is set to 25% of the value of the **-Xmx** option.

For both the **-Xgcpolicy:gencon** and **-Xgcpolicy:balanced** policies, the JVM returns an error if you try to use **-Xmnx** with **-Xmn**.

To find the value of **-Xmnx** that the JVM is using, specify the **-verbose:sizes** option on the command line.

Parameters

-Xmnx<*size*>

where <*size*> is an absolute value.

-Xms

The **-Xms** option sets the initial size of the Java heap. You can also use the **-Xmo** option.

Purpose

By using the **-Xms** option and the **-Xmx** option, you can control the size of the Java heap. The value of **-Xmx** must be greater than or equal to **-Xms**. For more information about **-Xmx**, see "-Xmx."

For information about default values, see "Default settings for the JVM" on page 171.

Parameters

-Xms<*size*>

where <*size*> is an absolute value. The minimum size is 1MB.

If scavenger is enabled, **-Xms** is greater than or equal to the sum value of **-Xmn** and **-Xmo**.

If scavenger is not enabled, **-Xms** is equal to the value of **-Xmo**.

-Xms*50m*

The Java heap starts at 50MB and grows to the maximum default value.

-Xms*20m* -Xmx*1024m*

The Java heap starts at 20MB and grows to a maximum size of 1GB.

-Xms*100m* -Xmx*100m*

The Java heap starts at 100MB and never grows.

-Xmx

The **-Xmx** option sets the maximum Java heap size.

Purpose

By using the **-Xmx** option and the **-Xms** option, you can control the size of the Java heap. The value of **-Xmx** must be greater than or equal to **-Xms**. For more information about **-Xms**, see "-Xms" on page 167.

For information about default values, see "Default settings for the JVM" on page 171.

If you are allocating the Java heap with large pages, read the information provided in the"-Xlp" on page 146 topic.

Parameters

-Xmx<*size*>

where <*size*> is an absolute value.

-Xmx*256m*

The Java heap starts at the default initial value and grows to a maximum of 256MB.

-Xms*20m* -Xmx*1024m*

The Java heap starts at 20MB and grows to a maximum size of 1GB.

-Xms*100m* -Xmx*100m*

The Java heap starts at 100MB and never grows.

-Xnuma

Use the **-Xnuma** option to turn off Non-Uniform Memory Architecture (NUMA) awareness when using the balanced garbage collection policy.

Purpose

By default, **-Xgcpolicy:balanced** uses features available on NUMA-enabled hardware and operating systems in order to improve application scalability. For more information about NUMA, see "NUMA awareness" on page 22. However, for workloads that do most of their work in one thread, or workloads that maintain a full heap, turning off NUMA awareness can improve performance.

Parameters

-Xnuma:none

Turns off NUMA awareness for the balanced garbage collection policy.

-Xsoftmx

This option sets the initial maximum size of the Java heap.

Purpose

When the initial maximum size of the Java heap is set, the GC attempts to shrink the heap to the new limit.

Parameters

-Xsoftmx<*size*>

To specify an initial Java heap size of 2GB, use **-Xsoftmx2g**.

Use the **-Xmx** option to set the maximum heap size. Use the com.ibm.lang.management API to alter the heap size limit between **-Xms** and **-Xmx** at run time. By default, this option is set to the same value as **-Xmx**.

When a lower **-Xsoftmx** value is set, the GC attempts to respect the new limit. However, the ability to shrink the heap depends on a number of factors. There is no guarantee that a decrease in the heap size will occur. If or when the heap shrinks below the new limit, the heap will not grow beyond that limit.

When the heap shrinks, the GC might release memory. The ability of the operating system to reclaim and use this memory varies based on the capabilities of the operating system.

Note: When using **-Xgcpolicy:gencon**, **-Xsoftmx** applies only to the non-nursery portion of the heap. In some cases the heap grows above the **-Xsoftmx** value because the nursery portion grows, pushing the heap size above the limit set. See **-Xmn** for limiting the nursery size.

-Xtgc

You can trace garbage collection operations with the **-Xtgc** option. The **-Xtgc:references** option is no longer available.

Purpose

The **-Xtgc** options turn on tracing that provides detailed information about garbage collection operations. For a list of options that are available with IBM SDK, Java Technology Edition, Version 6, see ../../../../com.ibm.java.doc.diagnostics.60/diag/tools/gcpd_tracing.html. The following section describes the changes to existing options.

Parameters

-Xtgc:references

This option, which in IBM SDK, Java Technology Edition, Version 6 shows activity relating to reference handling during garbage collections, is no longer available.

Balanced Garbage Collection policy options

The policy supports a number of command-line options to tune garbage collection (GC) operations.

About the policy

The policy uses a hybrid approach to garbage collection by targeting areas of the heap with the best return on investment. The policy tries to avoid global collections by matching allocation and survival rates. The policy uses mark, sweep, compact and generational style garbage collection. For more information about the Balanced Garbage Collection policy, see “Balanced Garbage Collection policy” on page 21. For information about when to use this policy, see “When to use the Balanced garbage collection policy” on page 25.

You specify the Balanced policy using the **-Xgcpolicy:balanced** command-line option. The following defaults apply:

Heap size

The initial heap size is *Xmx*/1024, rounded down to the nearest power of

2, where *Xmx* is the maximum heap size available. You can override this value by specifying the **-Xms** option on the command line.

Command-line options

The following options can also be specified on the command line with **-Xgcpolicy:balanced**:

- **-Xalwaysclassgc**
- **-Xclassgc**
- **-Xcompactexplicitgc**
- **-Xdisableexcessivegc**
- **-Xdisableexplicitgc**
- **-Xenableexcessivegc**
- **-Xgcthreads**<*number*>
- **-Xgcworkpackets**<*number*>
- **-Xmaxe**<*size*>
- **-Xmaxf**<*percentage*>
- **-Xmaxt**<*percentage*>
- **-Xmca**<*size*>
- **-Xmco**<*size*>
- **-Xmine**<*size*>
- **-Xminf**<*percentage*>
- **-Xmint**<*percentage*>
- **-Xmn**<*size*>
- **-Xmns**<*size*>
- **-Xmnx**<*size*>
- **-Xms**<*size*>
- **-Xmx**<*size*>
- **-Xnoclassgc**
- **-Xnocompactexplicitgc**
- **-Xnuma:**none
- **-Xsoftmx**<*size*>
- **-Xsoftrefthreshold**<*number*>
- **-Xverbosegclog***[:<file> [, <X>,<Y>]]*

A detailed description of these command line options can be found in ../../../../com.ibm.java.doc.diagnostics.60/diag/appendixes/cmdline/commands_gc.html.

The behavior of the following options is different when specified with **-Xgcpolicy:balanced**:

-Xcompactgc
Compaction occurs when a System.gc() call is received (default). Compaction always occurs on all other collection types.

-Xnocompactgc
Compaction does not occur when a System.gc() call is received. Compaction always occurs on all other collection types.

The following options are ignored when specified with **-Xgcpolicy:balanced**:

- **-Xconcurrentbackground**<*number*>
- **-Xconcurrentlevel**<*number*>
- **-Xconcurrentslack**<*size*>
- **-Xconmeter:**<*soa* | *loa* | *dynamic*>
- **-Xdisablestringconstantgc**
- **-Xenablestringconstantgc**
- **-Xgc:splitheap** (Windows 32-bit only)
- **-Xloa**
- **-Xloainitial**<*percentage*>
- **-Xloamaximum**<*percentage*>
- **-Xloaminimum**<*percentage*>
- **-Xmo**<*size*>
- **-Xmoi**<*size*>
- **-Xmos**<*size*>
- **-Xmr**<*size*>
- **-Xmrx**<*size*>
- **-Xnoloa**
- **-Xnopartialcompactgc**
- **-Xpartialcompactgc**

A detailed description of these command line options can be found in ../../../../com.ibm.java.doc.diagnostics.60/diag/appendixes/cmdline/commands_gc.html.

Default settings for the JVM

This appendix shows the default settings that the JVM uses. These settings affect how the JVM operates if you do not apply any changes to its environment. The tables show the JVM operation and the default setting.

These tables are a quick reference to the state of the JVM when the JVM is first installed. The last column shows how the default setting can be changed:

c The setting is controlled by a command-line parameter only.

e The setting is controlled by an environment variable only.

ec The setting is controlled by a command-line parameter or an environment variable. The command-line parameter always takes precedence.

JVM setting	Default	Setting affected by
Javadump	Enabled	ec
Heapdump	Disabled	ec
System dump	Enabled	ec
Snap traces	Enabled	ec
Verbose output	Disabled	c
Boot classpath search	Disabled	c
JNI checks	Disabled	c
Remote debugging	Disabled	c

JVM setting	Default	Setting affected by
Strict conformance checks	Disabled	c
Quickstart	Disabled	c
Remote debug info server	Disabled	c
Reduced signaling	Disabled	c
Signal handler chaining	Enabled	c
Classpath	Not set	ec
Class data sharing	Disabled	c
Accessibility support	Enabled	e
JIT compiler	Enabled	ec
AOT compiler (AOT is not used by the JVM unless shared classes are also enabled)	Enabled	c
JIT debug options	Disabled	c
Java2D max size of fonts with algorithmic bold	14 point	e
Java2D use rendered bitmaps in scalable fonts	Enabled	e
Java2D freetype font rasterizing	Enabled	e
Java2D use AWT fonts	Disabled	e

JVM setting	AIX	IBM i	Linux	Windows	z/OS	Setting affected by
Default locale	None	None	None	N/A	None	e
Time to wait before starting plug-in	N/A	N/A	Zero	N/A	N/A	e
Temporary directory	`/tmp`	`/tmp`	`/tmp`	`c:\temp`	`/tmp`	e
Plug-in redirection	None	None	None	N/A	None	e
IM switching	Disabled	Disabled	Disabled	N/A	Disabled	e
IM modifiers	Disabled	Disabled	Disabled	N/A	Disabled	e
Thread model	N/A	N/A	N/A	N/A	Native	e
Initial stack size for Java Threads 32-bit. Use: **-Xiss<size>**	2 KB	2 KB	2 KB	2 KB	2 KB	c
Maximum stack size for Java Threads 32-bit. Use: **-Xss<size>**	256 KB	256 KB	256 KB	256 KB	256 KB	c
Stack size for OS Threads 32-bit. Use **-Xmso<size>**	256 KB	256 KB	256 KB	32 KB	256 KB	c
Initial stack size for Java Threads 64-bit. Use: **-Xiss<size>**	2 KB	N/A	2 KB	2 KB	2 KB	c
Maximum stack size for Java Threads 64-bit. Use: **-Xss<size>**	512 KB	N/A	512 KB	512 KB	512 KB	c
Stack size for OS Threads 64-bit. Use **-Xmso<size>**	256 KB	N/A	256 KB	256 KB	256 KB	c
Initial heap size. Use **-Xms<size>**	4 MB	4 MB	4 MB	4 MB	4 MB	c

JVM setting	AIX	IBM i	Linux	Windows	z/OS	Setting affected by
Maximum Java heap size. Use **-Xmx<size>**	Half the available memory with a minimum of 16 MB and a maximum of 512 MB	2 GB	Half the available memory with a minimum of 16 MB and a maximum of 512 MB	Half the available memory with a minimum of 16 MB and a maximum of 512 MB. See note.	Half the available memory with a minimum of 16 MB and a maximum of 512 MB	c

Note: This change is specific to the IBM J9 2.6 virtual machine. For versions of the IBM SDK, Java Technology Edition, Version 6 that contain an IBM J9 2.4 virtual machine, the value of **-Xmx** for the Windows JVM is half the physical memory. The minimum value is 16 MB and the maximum value is 2 GB.

"Available memory" is defined as being the smallest of two values:

- The real or "physical" memory.
- The **RLIMIT_AS** value.

Known issues and limitations

Known issues and limitations.

IBM SDK for Linux fails on Red Hat Enterprise Linux (RHEL) V4

When running the IBM J9 2.6 virtual machine on RHEL 4, the JVM fails, generating a core dump. The failure occurs because this version of the JVM is not supported on RHEL 4. For information about the supported operating systems, see Chapter 4, "Hardware and software requirements," on page 31.

Chinese, Japanese, or Korean characters are not displayed properly in GUI applications on RHEL 6

This problem occurs when using the Motif AWT. The problem has the effect that Chinese, Japanese, or Korean characters are not displayed properly in GUI applications.

The workaround is to use XAWT instead of Motif AWT.

Position for ibus composition window is incorrect on RHEL 6

This problem occurs when using the ibus input method. The effect is that the Input Method Editor (IME) composition window is not displayed under the cursor position. An additional effect is that the composition window does not follow the xterm window if it is moved.

This problem only affects IBM POWER and s390 platforms.

If you encounter this problem, contact Red Hat for further information.

IBM SDK for Linux fails on SUSE Linux Enterprise Server (SLES) V9

When running the IBM J9 2.6 virtual machine on SLES 9, the JVM fails, generating a core dump. The failure occurs because this version of the JVM is not supported on SLES 9. For information about the supported operating systems, see Chapter 4, "Hardware and software requirements," on page 31.

IBM SDK for Windows fails on Windows 2000 Server

When running the IBM J9 2.6 virtual machine on Windows 2000 server, the JVM fails, generating a core dump. The failure occurs because this version of the JVM is not supported on Windows 2000 server. For information about the supported operating systems, see Chapter 4, "Hardware and software requirements," on page 31.

The methods `setReadOnly()` and `setWritable(false)` do not work on Windows directories

From Service Refresh 1, if you use these methods on a directory on the Windows operating system, they return the value `false`.

Note: In the same situation in earlier releases, these methods set the DOS read-only attribute to prevent the directory from being deleted. However, this behaviour does not make the directory read-only, therefore the only changes in behavior are that the methods now return the value `false`, and the read-only attribute is not set.

Chinese characters stored as ? in an Oracle database

When you configure an Oracle database to use the ZHS16GBK character set, some Chinese characters or symbols that are encoded with the GBK character set are incorrectly stored as a question mark (?). This problem is caused by an incompatibility of the GBK undefined code range Unicode mapping between the Oracle ZHS16GBK character set and the IBM GBK converter. To fix this problem, use a new code page, MS936A, by including the following system property when you start the JVM:

```
-Dfile.encoding=MS936A
```

For IBM WebSphere Application Server users, this problem might occur when web applications that use JDBC configure Oracle as the WebSphere Application Server data source. To fix this problem, use a new code page, MS936A, as follows:

1. Use the following system property when you start the JVM:

   ```
   -Dfile.encoding=MS936A
   ```

2. Add the following lines to the *WAS_HOME*`/properties/converter.properties` file, where *WAS_HOME* is your WebSphere Application Server installation directory.

   ```
   GBK=MS936A
   GB2312=MS936A
   ```

Incorrect value for Windows 8.1 and Windows 10 in the `java -version` output

The executable files in this release do not contain the manifest information that is required to properly display the Windows version information in the output from the **`java -version`** command. Windows 8.1 and Windows 10 are incorrectly

reported as Windows 8.

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

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 jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those

websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide 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:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

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.

The performance data discussed herein is presented as derived under specific operating conditions. Actual results may vary.

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.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year).
Portions of this code are derived from IBM Corp. Sample Programs.
© Copyright IBM Corp. _enter the year or years_.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Intel, Intel logo, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see: (i) IBM’s Privacy Policy at http://www.ibm.com/privacy ; (ii) IBM’s Online Privacy Statement at http://www.ibm.com/privacy/details (in particular, the section entitled “Cookies, Web Beacons and Other Technologies”); and (iii) the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.

Index

Special characters

A

B

C

D

E

G

X

Z

Printed in USA