IBM SDK for z/OS, Java Technology Edition
Version 7

SDK and Runtime Guide

IBM SDK for z/OS, Java Technology Edition
Version 7

SDK and Runtime Guide

Note
Before using this information and the product it supports, read the information in "Notices" on page 481.

Contents

Preface

This user guide provides general information about the IBM® SDK for z/OS®, Java™ Technology Edition, Version 7. The guide gives specific information about any differences in the IBM implementation compared with the Oracle implementation.

This user guide applies to IBM SDK for z/OS, Java Technology Edition, Version 7.

Read this user guide with the more extensive documentation on the Oracle Web site: http://www.oracle.com/technetwork/java/index.html.

The terms *Runtime Environment* and *Java Virtual Machine* are used interchangeably throughout this user guide.

The Program Code is not designed or intended for use in real-time applications such as (but not limited to) the online control of aircraft, air traffic, aircraft navigation, or aircraft communications; or in the design, construction, operation, or maintenance of any nuclear facility.

Chapter 1. Product overview

Gain a quick understanding of the product, its new features, and conventions that are used elsewhere in this documentation.

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

Late breaking information about the IBM SDK for z/OS, V7 that is not available in the user guide can be found here: http://www.ibm.com/support/docview.wss?uid=swg21499721

Introduction to Java

The IBM implementation of the Java platform provides a development toolkit and an application runtime environment.

The Java programming language is a high-level, object-oriented language. When written, a Java program is compiled into *bytecode*. The *bytecode* is interpreted at run time by a platform-specific Java component. This component acts as a translator between the language and the underlying operating system and hardware. This staged approach to compiling and interpreting Java applications, means that application code can be easily ported across hardware platforms and operating systems.

The IBM implementation of the Java platform is based upon the Java Technology developed by Oracle Corporation. IBM supply two installable packages depending on platform: the Software Developers Kit (SDK) and the Java Runtime Environment (JRE). The key components in these packages are detailed in the following sections.

IBM Software Developers Kit (SDK)

The SDK contains development tools and a Java Runtime Environment (JRE).

The SDK is an installable Java package, which contains the Java Application Programming Interface (API). The Java API is a large collection of ready-made classes, grouped into libraries, that help you develop and deploy applications. The SDK also includes:

- The Java Compiler.
- The IBM Java Runtime Environment (JRE) and IBM Java Virtual machine (JVM).
- Tools for monitoring, debugging, and documenting applications.
- Tools for developing user interfaces, or GUIs.
- Integration libraries for applications that must access databases and remote objects.

The SDK package contains a *readme* file that provides links to the online information center documentation, and to downloadable documentation. The documentation available for download includes this guide, in multiple formats.

When the package is installed, the SDK tools can be found in the `/usr/lpp/java/J7.0[_64]/bin` directory.

Applications written entirely in Java must have **no** dependencies on the IBM SDK directory structure (or files in those directories). Any dependency on the SDK directory structure (or the files in those directories) might result in application portability problems.

Contents of the SDK

SDK tools:

appletviewer (Java Applet Viewer)
Tests and runs applets outside a web browser.

apt (Annotation Processing Tool)
Finds and runs annotation processors based on the annotations present in the set of specified source files being examined.This tool is deprecated. See http://docs.oracle.com/javase/7/docs/technotes/guides/apt/index.html.

extcheck (Extcheck utility)
Detects version conflicts between a target jar file and jar files that are currently installed.

hwkeytool
Manages a keystore of private keys and their associated X.509 certificate chains authenticating the corresponding public keys.

idlj (IDL to Java Compiler)
Generates Java bindings from a given IDL file.

jar (Java Archive Tool)
Combines multiple files into a single Java Archive (JAR) file.

jarsigner (JAR Signing and Verification Tool)
Generates signatures for JAR files and verifies the signatures of signed JAR files.

java (Java Interpreter)
Runs Java classes. The Java Interpreter runs programs that are written in the Java programming language.

java-rmi.cgi (HTTP-to-CGI request forward tool)
Accepts RMI-over-HTTP requests and forwards them to an RMI server listening on any port.

javac (Java Compiler)
Compiles programs that are written in the Java programming language into bytecodes (compiled Java code).

javadoc (Java Documentation Generator)
A utility to generate HTML pages of API documentation from Java source files.

javah (C Header and Stub File Generator)
Enables you to associate native methods with code written in the Java programming language.

javap (Class File Disassembler)
Disassembles compiled files and can print a representation of the bytecodes.

javaw (Java Interpreter)
Runs Java classes in the same way as the **java** command does, but does not use a console window.

jconsole (JConsole Monitoring and Management Tool)
Monitors local and remote JVMs using a GUI. JMX-compliant.

jdmpview (Cross-platform dump formatter)
Analyzes dumps. For more information, see the "Problem determination" on page 173.

keytool (Key and Certificate Management Tool)
Manages a keystore (database) of private keys and their associated X.509 certificate chains that authenticate the corresponding public keys.

native2ascii (Native-To-ASCII Converter)
Converts a native encoding file to an ASCII file that contains characters encoded in either Latin-1 or Unicode, or both.

policytool (Policy File Creation and Management Tool)
Creates and modifies the external policy configuration files that define the Java security policy for your installation.

rmic (Java Remote Method Invocation (RMI) Stub Converter)
Generates stubs, skeletons, and ties for remote objects. Includes RMI over Internet Inter-ORB Protocol (RMI-IIOP) support.

rmid (RMI activation system daemon)
Starts the activation system daemon so that objects can be registered and activated in a Java Virtual Machine (JVM).

rmiregistry (Java remote object registry)
Creates and starts a remote object registry on the specified port of the current host.

schemagen
Creates a schema file for each namespace referenced in your Java classes.

serialver (Serial Version Command)
Returns the serialVersionUID for one or more classes in a format that is suitable for copying into an evolving class.

tnameserv (Common Object Request Broker Architecture (CORBA) transient naming service)
Starts the CORBA transient naming service.

wsgen
Generates JAX-WS portable artifacts used in JAX-WS Web services.

wsimport
Generates JAX-WS portable artifacts from a Web Services Description Language (WSDL) file.

xjc
Compiles XML Schema files.

z/OS batch toolkit
A set of tools that enhances Java batch capabilities and use of system interfaces on z/OS. The toolkit includes:

- A native launcher for running Java applications directly as batch jobs or started tasks.
- A set of Java classes that make access to traditional z/OS data and key system services directly available from Java applications.
- Console communication, multiline WTO (write to operator), and return code passing capability.

For more information about the z/OS batch toolkit, see http://www.ibm.com/systems/z/os/zos/tools/java/products/jzos/overview.html.

Include Files
C headers for JNI programs.

Demos
The `demo` directory contains a number of subdirectories containing sample source code, demos, applications, and applets that you can use.

readme file
A text file containing minimal information about how to get started. This file provides links to online and downloadable documentation, including Java API documentation for the IBM SDK.

Copyright notice
The copyright notice for the SDK for z/OS software.

IBM Java Runtime Environment (JRE)

The JRE provides runtime support for Java applications.

The JRE includes the IBM Java Virtual Machine (JVM), which interprets Java *bytecode* at run time. There are a number of tools included with the JRE that are installed into the `/usr/lpp/java/J7.0[_64]/jre/bin` directory, unless otherwise specified.

Contents of the JRE

Core Classes
These classes are the compiled class files for the platform and must remain compressed for the compiler and interpreter to access them. Do not modify these classes; instead, create subclasses and override where you need to.

Trusted root certificates
From certificate signing authorities. These certificates are used to validate the identity of signed material.

JRE Tools

ikeycmd (iKeyman command-line utility)
Allows you to manage keys, certificates, and certificate requests from the command line. For more information see the accompanying Security documentation.

ikeyman (iKeyman GUI utility)
Allows you to manage keys, certificates, and certificate requests. For more information see the accompanying Security documentation, which includes the *iKeyman User Guide*. There is also a command-line version of this utility.

Note: The GUI version of this utility is not supported on the z/OS operating system.

java (Java Interpreter)
Runs Java classes. The Java Interpreter runs programs that are written in the Java programming language.

javaw (Java Interpreter)
Runs Java classes in the same way as the **java** command does, but does not use a console window.

jextract (Dump extractor)
Converts a system-produced dump into a common format that can be used by **jdmpview**. For more information, see "Using jextract" on page 276.

keytool (Key and Certificate Management Tool)
Manages a keystore (database) of private keys and their associated X.509 certificate chains that authenticate the corresponding public keys.

kinit
Obtains and caches Kerberos ticket-granting tickets.

klist
Displays entries in the local credentials cache and key table.

ktab
Manages the principal names and service keys stored in a local key table.

pack200
Transforms a JAR file into a compressed pack200 file using the Java gzip compressor.

policytool (Policy File Creation and Management Tool)
Creates and modifies the external policy configuration files that define the Java security policy for your installation.

rmid (RMI activation system daemon)
Starts the activation system daemon so that objects can be registered and activated in a Java Virtual Machine (JVM).

rmiregistry (Java remote object registry)
Creates and starts a remote object registry on the specified port of the current host.

tnameserv (Common Object Request Broker Architecture (CORBA) transient naming service)
Starts the CORBA transient naming service.

unpack200
Transforms a packed file produced by **pack200** into a JAR file.

IBM Java Virtual Machine (JVM)

The IBM Java Virtual machine (JVM) is the platform-specific component that runs a Java program.

At run time, the JVM interprets the Java bytecode that has been compiled by the Java Compiler. The JVM acts as a translator between the language and the underlying operating system and hardware. A Java program requires a specific JVM to run on a particular platform, such as Linux, z/OS, or Windows.

The main components of the IBM JVM are:

- JVM Application Programming Interface (API)
- Diagnostic component
- Memory management
- Class loader
- Interpreter
- Platform port layer

For further information about the JVM, see "Components of the IBM Virtual Machine for Java" on page 21.

Different versions of the IBM SDK contain different implementations of the JVM. You can identify the implementation in the output from the `java -version` command, which gives these strings for the different implementations:

Implementation	Output
7	`IBM J9 VM (build 2.6, JRE 1.7.0 ...`
6	`IBM J9 VM (build 2.4, JRE 1.6.0 IBM...` `IBM J9 VM (build 2.6, JRE 1.6.0 ...`
5.0	`IBM J9 VM (build 2.3, J2RE 1.5.0 IBM...`
1.4.2 'classic'	`Classic VM (build 1.4.2, J2RE 1.4.2 IBM...`
1.4.2 on z/OS 64-bit and AMD64/EM64T platforms	`IBM J9SE VM (build 2.2, J2RE 1.4.2 IBM...`

What's new

Learn about the new features and functions available with IBM SDK for z/OS, V7.

This topic introduces new material for Version 7. On the z/OS platform, some of these functions were first introduced in the 31-bit and 64-bit releases of IBM SDK for z/OS, Java Technology Edition V6.0.1.

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

General changes are listed in this topic. For changes that relate to specific components, see the relevant subtopic.

New path in font configuration properties file

From z/OS V2.1, fonts are provided by the operating system. The paths to the font files in the *lib_dir*`/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 "Using non-default system fonts" on page 111.

IBM J9 Virtual Machine

IBM SDK for z/OS, V7 includes the IBM J9 V2.6 virtual machine (JVM). Read about new features and capabilities that are included with this JVM.

The sun.reflect.Reflection.getCallerClass(int depth) method is no longer supported

To enhance security, the sun.reflect.Reflection.getCallerClass(int depth) method is no longer supported. Use the sun.reflect.Reflection.getCallerClass() method instead. This method always uses a depth of 2.

If you use the sun.reflect.Reflection.getCallerClass(int depth) method in your application, an UnsupportedOperationException exception is thrown.

You can re-enable support for this method by using the "-Djdk.reflect.allowGetCallerClass" on page 423 system property, however this property will be removed in a future release.

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, see "Securing JAXP processing against malformed input" on page 117.

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" on page 449.

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" on page 420.

LiveConnect support is now disabled in certain circumstances

In previous releases, LiveConnect (a browser feature that enables you to use Java APIs from within JavaScript code) was always enabled. From this release, LiveConnect is disabled in the following circumstances:

- The security level on the **Security** tab of the IBM Control Panel for Java is set to **Very High**.
- The security level on the **Security** tab of the IBM Control Panel for Java is set to **High** (default) and the runtime environment has expired. The runtime environment automatically expires after 6 months. To ensure that your runtime environment remains secure, install the latest release.

Note: The security level setting is present only in very recent releases.

Behavior change to java.lang.logging.Logger

To enhance security, java.lang.logging.Logger no longer walks the stack to search for resource bundles. Instead, the resource bundles are located by using the caller 's class loader. If your application depends upon stack-walking to locate resource bundles, this behavior change might affect your application. To work around this problem, a system property is available in this release to revert to the earlier behavior. To set this property on the command line specify: **`-Djdk.logging.allowStackWalkSearch=true`**.

Support for 2GB large pages

On z/OS V1.13 with the RSM Enablement Offering, you can now request the JVM to allocate the Java object heap with 2GB nonpageable large pages, 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 "Configuring large page memory allocation" on page 154. For more information about the RSM Enablement Offering, see http://www-03.ibm.com/systems/z/os/zos/downloads/#RSME.

Use of large pages by default

On certain platforms and processors, the JVM now starts with large pages enabled by default for both the JIT codecache and the objectheap, instead of the default operating system page size.

On z/OS, the JVM uses 1M pageable large pages, when running on the 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.

Note: **PAGESCM=ALL | NONE** in the IEASYSxx parmlib member controls pageable 1M large pages for the entire LPAR. ALL is the default.

If the operating system is not configured for the large page size, or if the correct hardware is not available, the JVM uses the default operating system page size instead.

You can configure the large page size using the **-Xlp** command. 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** command options, see "JVM command-line options" on page 428.

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 "JVM command-line options" on page 428.

Enabling caching of LUDCL

A new system property is available to enable caching of the Latest User Defined Class Loader (LUDCL). By reducing repeated lookups, Java applications that use deserialization extensively can see a performance improvement. For more information see **-Dcom.ibm.enableClassCaching** in "System property command-line options" on page 419.

Use of java.util.* package and classes

The SDK now uses the Oracle implementation of the java.util.* package, including all classes within the package. Earlier releases of the SDK used customized versions of the Apache Harmony class libraries. This change establishes a common implementation point for the java.util.* package, enabling consistent performance and behavior characteristics across Java implementations. Existing applications are expected to function without problems. However, if testing exposes any issues, contact your IBM service representative.

IBM GBK converter

By default the IBM GBK converter follows Unicode 3.0 standards. A new system property value for **-Dfile.encoding** is available to force the IBM GBK converter to follow Unicode 2.0 standards. For more information see "System property command-line options" on page 419.

IBM z/OS Language Environment®

JVM signal handlers for SIGSEGV, SIGILL, SIGBUS, SIGFPE, SIGTRAP, and for SIGABRT by default terminate the process using exit(). If you are using the IBM z/OS Language Environment (LE), LE 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 "JVM command-line options" on page 428.

Java Attach API support is disabled by default

To enhance security on z/OS, support for the Java Attach API is now disabled by default. For more information, see "Support for the Java Attach API" on page 141.

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. However, 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, and the z/OS V1.13 Remote Storage Manager Enablement Offering web deliverable.

For more information, see the **-Xlp** option in "JVM command-line options" on page 428.

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. For more information about the system property, see "System property command-line options" on page 419.

An enhanced hashing algorithm is 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, see “System property command-line options” on page 419.

Compressed references tuning option

A new command-line option is available to override the allocation strategy used by the 64-bit JVM when running with compressed references enabled. The **-XXnosuballoc32bitmem** option prevents the JVM pre-allocating an area of virtual memory, leaving the operating system to handle the allocation strategy. For more information, see “JVM -XX command-line options” on page 446.

Changes to locale translation files

Changes are made to the locale translation files to make them consistent with Oracle JDK 7. The same changes were also applied to the IBM SDK for Java V6 for consistency with Oracle JDK 6. To understand the differences in detail, see this support document for Java 6: http://www.ibm.com/support/docview.wss?uid=swg21568667.

JVM optimizations

The IBM J9 V2.6 virtual machine includes new optimizations for Java 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 “Testing JVM optimizations” on page 192.

Java Attach API

Connections to virtual machines through the Java Attach API have a new default state. By default, the Attach API is enabled on ALL platforms. The exception is that for security reasons, processes on z/OS using the default z/OS OMVS segment cannot enable the Attach API. For more information, see “Support for the Java Attach API” on page 141.

Deprecation of JRIO in IBM SDK for Java 7 on z/OS platform

The JRIO component available in earlier versions of the IBM SDK for Java has been supplanted by the increasing functionality and enhancements of the JZOS component.

In IBM SDK for Java 7 on z/OS, 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 http://www.ibm.com/systems/z/os/zos/tools/java/products/jzos/overview.html. 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 the Java z/OS website.

Memory management

Changes to memory management for IBM SDK for z/OS, V7 including garbage collection policies, new command-line options, and verbose garbage collection logging.

Configuring the initial maximum Java heap size

The **-Xsoftmx** option is now available on Linux, Windows, and z/OS, as well as AIX®. A soft limit for the maximum heap size can be set 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 "Garbage Collector command-line options" on page 453.

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 248.

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 403.

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 39.

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 options" on page 151.

Verbose 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 334.

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 455.

Class data sharing

Class data sharing provides a method of reducing memory footprint and improving JVM startup time. Read about changes to class data sharing that are introduced in IBM SDK for z/OS, V7.

The following sections describe the changes.

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 353. For more information about the **-Xshareclasses** suboptions **enableBCI** and **rcdSize**, see "Class data sharing command-line options" on page 159.

.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. For more information, see the **-Xzero** option in "JVM command-line options" on page 428.

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 347.

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 347.

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 “Dump agents” on page 224.

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 “Obtaining information about shared caches” on page 360.

New IBM JVMTI extensions are included, that can search for shared class caches, and remove a shared class cache. For more information, see “IBM JVMTI extensions - API reference” on page 392.

printStats utility

The printStats utility shows more information about the line number status of classes in the shared cache, as well as information about other new items, such as JIT data and class debug area.

You can use parameters with the printStats utility, to get more detailed information about specific types of cache content.

For more information, see “printStats utility” on page 362.

Controlling shared cache directory permissions

You can use the **-Xshareclasses:cacheDirPerm** command-line option to control the permissions of directories that are created for shared caches. For more information, see “JVM command-line options” on page 428.

The JIT compiler

A new feature is available that improves the performance of 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 “How the JIT compiler optimizes code” on page 58.

Diagnostic component

There are continuous improvements for diagnosing problems with the IBM J9 virtual machine, including improved logging and analysis of native memory.

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 251.

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 243 for an example.

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 246.

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 251.

Working with system dumps containing multiple JVMs

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 277.

Using the dump viewer with compressed files, or in batch mode

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 274.

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 278.

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 235.

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 243.

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 252.

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 228.

Receiving OutofMemoryError exceptions

From service refresh 1, an error message is generated when there is an OutofMemoryError condition on the Java heap.

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 290.

Diagnosing problems with native memory

Information about native memory usage is now provided by a Java dump. For further information, including example output, see "Native memory (NATIVEMEMINFO)" on page 246.

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 Diagnostic Tool Framework for Java" on page 405.

In addition, you can query native memory usage by using a new IBM JVMTI extension. The GetMemoryCategories() API returns the JRE native memory use by memory category. For further information about the IBM JVMTI extension, see "Querying JRE native memory categories" on page 397.

Diagnosing problems with blocked threads

The THREADS section of the Java dump output now contains information about the resources that blocked threads are waiting for. For more information, see "Blocked thread information" on page 257.

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. For more information about message logging, see the **-Xlog** option in "JVM command-line options" on page 428.

You can also query and modify the message setting by using new IBM JVMTI extensions. For more information about these JVMTI extensions, see "IBM JVMTI extensions" on page 390.

Processing system dumps

The **jextract** utility performs some important steps in the diagnostic process for earlier versions of the SDK. These steps have now been automated to simplify the process and make reporting a problem to IBM support much easier. For more information about viewing system dumps, see "Using the dump viewer" on page 273.

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 234. If you want to disable this new functionality, remove the new dump agent. For more information, see "Removing dump agents" on page 235.

Portable Heap Dump (PHD) file format

Detailed information about Portable Heap Dump (PHD) file formats are provided to assist with problem diagnosis. For more information, see "Portable Heap Dump (PHD) file format" on page 266.

Conventions and terminology

Specific conventions are used to describe methods and classes, and command-line options.

Methods and classes are shown in normal font:

- The serviceCall() method
- The StreamRemoteCall class

Command-line options are shown in bold. For example:

- **-Xgcthreads**

Options shown with values in braces signify that one of the values must be chosen. For example:

-Xverify:{remote | all | none}

with the default underscored.

Options shown with values in brackets signify that the values are optional. For example:

-Xrunhprof[:**help**][<*suboption*>=<*value*>...]

In this information, any reference to Oracle is intended as a reference to Oracle Corporation.

Directory conventions

In the following directories, <*version*> is a single-digit version number that represents the product version, and <*release*> is a single-digit version number that represents the product release.

install_dir

The installation directory is referred to as *install_dir* in this documentation. The default installation directory is as follows:

- /usr/lpp/java/J<*version*>.<*release*>[_64]/

For example:

- /usr/lpp/java/J7.0_64/

lib_dir

The Java library directory is referred to as *lib_dir* in this documentation. The library directory is as follows:

- *install_dir*/jre/lib/<*arch*>/

Java virtual machine (JVM) version conventions

The JVM version is referred to as <*vm_version*> in this documentation. To find out which version of the JVM you are using, enter the following command:

```
java -version
```

The following example output shows the JVM version in bold text, in the line beginning with IBM J9 VM:

```
java version "1.7.0"
Java(TM) SE Runtime Environment (build pxi3270sr1-20120201_02(SR1))
IBM J9 VM (build 2.6, JRE 1.7.0 Linux x86-32 20120131_101270 (JIT enabled, AOT enabled)
J9VM - R26_JVM_26_20120125_1726_B100726
JIT - r11_20120130_22318
GC - R26_JVM_26_20120125_1044_B100654
J9CL - 20120131_101270)
JCL - 20120127_01 based on Oracle 7u3-b02
```

The format of <*vm_version*> is digits only, so in the previous example, <*vm_version*> is 26.

Other sources of information

You can obtain additional information about the latest tools, Java documentation, and the IBM SDKs by following the links.

- For the IBM SDKs, see the downloads at:

  http://www.ibm.com/developerworks/java/jdk/index.html
- To download IBM SDK documentation as an Eclipse plug-in, or in PDF format for printing, see:

  http://www.ibm.com/developerworks/java/jdk/docs.html
- For any late breaking information that is not in this guide, see:

  http://www.ibm.com/support/docview.wss?uid=swg21499721
- For Javadoc HTML documentation that has been generated from IBM SDK for Java APIs, see:

  API documentation
- For articles, tutorials and other technical resources about Java Technology, see IBM developerWorks® at:

  http://www.ibm.com/developerworks/java/
- For Java documentation produced by Oracle, see:

http://www.oracle.com/technetwork/java/index.html

Accessibility

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

IBM strives to provide products with usable access for everyone, regardless of age or ability.

For example, you can operate the IBM SDK for z/OS, V7 without a mouse, by using only the keyboard.

Issues that affect accessibility are included in "Known issues and limitations" on page 476.

Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

For users who require keyboard navigation, a description of useful keystrokes for Swing applications can be found here: Swing Key Bindings.

IBM and accessibility

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility.

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

The information in this section of the *Information Center* provides a basic understanding of SDK components.

The content provides:

- Background information to explain why some diagnostic tools work the way they do
- Useful information for application designers
- An explanation of some parts of the JVM
- A set of topics on Garbage collection techniques, which are typically complex

Other sections provide a summary, especially where guidelines about the use of the SDK are appropriate. This content is not intended as a description of the design of the SDK, except that it might influence application design or promote an understanding of why things are done the way that they are.

A section that describes the IBM Object Request Broker (ORB) component is also available.

The sections in this part are:

The building blocks of the IBM Virtual Machine for Java

The IBM Virtual Machine for Java (JVM) is a core component of the Java Runtime Environment (JRE) from IBM. The JVM is a virtualized computing machine that follows a well-defined specification for the runtime requirements of the Java programming language.

The JVM is called "virtual" because it provides a machine interface that does not depend on the underlying operating system and machine hardware architecture. This independence from hardware and operating system is a cornerstone of the write-once run-anywhere value of Java programs. Java programs are compiled into "bytecodes" that target the abstract virtual machine; the JVM is responsible for executing the bytecodes on the specific operating system and hardware combinations.

The JVM specification also defines several other runtime characteristics.

All JVMs:

- Execute code that is defined by a standard known as the class file format
- Provide fundamental runtime security such as bytecode verification
- Provide intrinsic operations such as performing arithmetic and allocating new objects

JVMs that implement the specification completely and correctly are called "compliant". The IBM Virtual Machine for Java is certified as compliant. Not all compliant JVMs are identical. JVM implementers have a wide degree of freedom to define characteristics that are beyond the scope of the specification. For example, implementers might choose to favour performance or memory footprint; they might design the JVM for rapid deployment on new platforms or for various degrees of serviceability.

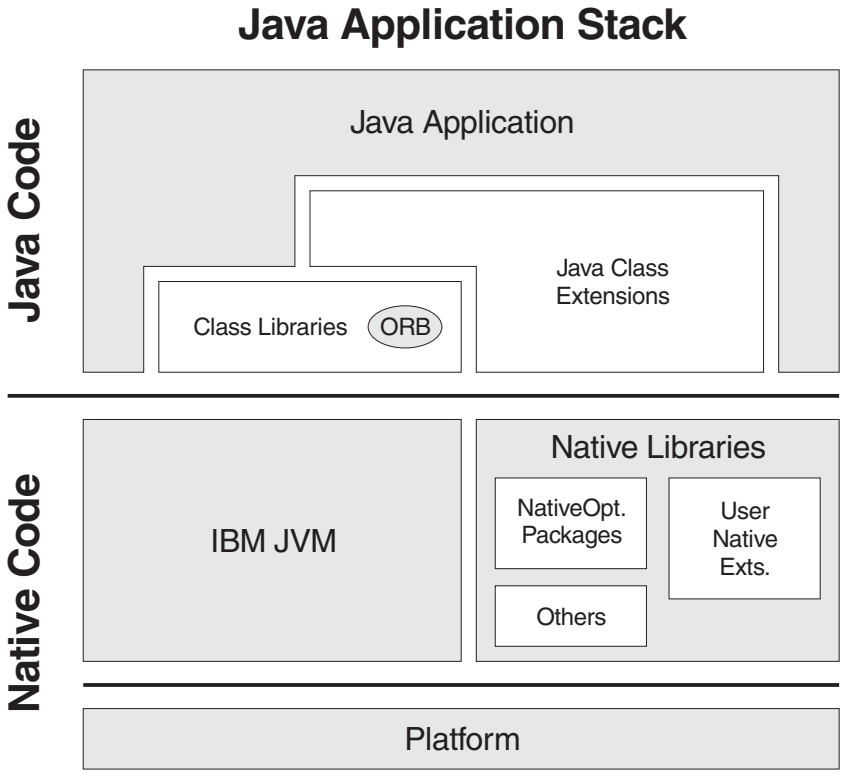
All the JVMs that are currently used commercially come with a supplementary compiler that takes bytecodes and produces platform-dependent machine code. This compiler works with the JVM to select parts of the Java program that could benefit from the compilation of bytecode, and replaces the JVM's virtualized interpretation of these areas of bytecode with concrete code. This is called just-in-time (JIT) compilation. IBM's JIT compiler is described in "The JIT compiler" on page 57.

The diagnostic information in this guide discusses the characteristics of the IBM JRE that might affect the non-functional behavior of your Java program. This guide also provides information to assist you with tracking down problems and offers advice, from the point of view of the JVM implementer, on how you can tune your applications. There are many other sources for good advice about Java performance, descriptions of the semantics of the Java runtime libraries, and tools to profile and analyze in detail the execution of applications.

Java application stack

A Java application uses the Java class libraries that are provided by the JRE to implement the application-specific logic. The class libraries, in turn, are implemented in terms of other class libraries and, eventually, in terms of primitive native operations that are provided directly by the JVM. In addition, some applications must access native code directly.

The following diagram shows the components of a typical Java Application Stack and the IBM JRE.

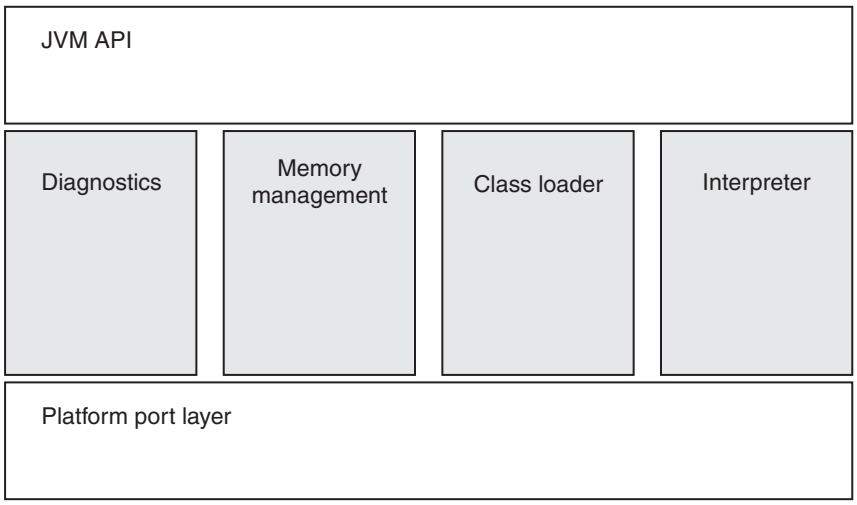


The JVM facilitates the invocation of native functions by Java applications and a number of well-defined Java Native Interface functions for manipulating Java from native code (for more information, see "The Java Native Interface (JNI)" on page 88).

Components of the IBM Virtual Machine for Java

The IBM Virtual Machine for Java technology comprises a set of components.

The following diagram shows component structure of the IBM Virtual Machine for Java:



JVM Application Programming Interface (API)

The JVM API encapsulates all the interaction between external programs and the JVM.

Examples of this interaction include:

- Creation and initialization of the JVM through the invocation APIs.
- Interaction with the standard Java launchers, including handling command-line directives.
- Presentation of public JVM APIs such as JNI and JVMTI.
- Presentation and implementation of private JVM APIs used by core Java classes.

Diagnostic component

The diagnostic component provides Reliability, Availability, and Serviceability (RAS) facilities to the JVM.

The IBM Virtual Machine for Java is distinguished by its extensive RAS capabilities. The JVM is designed to be deployed in business-critical operations and includes several trace and debug utilities to assist with problem determination.

If a problem occurs in the field, it is possible to use the capabilities of the diagnostic component to trace the runtime function of the JVM and help to identify the cause of the problem. The diagnostic component can produce output selectively from various parts of the JVM and the JIT. “Using diagnostic tools” on page 212 describes various uses of the diagnostic component.

Memory management

The memory management component is responsible for the efficient use of system memory by a Java application.

Java programs run in a managed execution environment. When a Java program requires storage, the memory management component allocates the application a discrete region of unused memory. After the application no longer refers to the storage, the memory management component must recognize that the storage is unused and reclaim the memory for subsequent reuse by the application or return it to the operating system.

The memory management component has several policy options that you can specify when you deploy the application. “Memory management” on page 23 discusses memory management in the IBM Virtual Machine for Java.

Class loader

The class loader component is responsible for supporting Java's dynamic code loading facilities.

The dynamic code loading facilities include:

- Reading standard Java .class files.
- Resolving class definitions in the context of the current runtime environment.
- Verifying the bytecodes defined by the class file to determine whether the bytecodes are language-legal.
- Initializing the class definition after it is accepted into the managed runtime environment.
- Various reflection APIs for introspection on the class and its defined members.

Interpreter

The interpreter is the implementation of the stack-based bytecode machine that is defined in the JVM specification. Each bytecode affects the state of the machine and, as a whole, the bytecodes define the logic of the application.

The interpreter executes bytecodes on the operand stack, calls native functions, contains and defines the interface to the JIT compiler, and provides support for intrinsic operations such as arithmetic and the creation of new instances of Java classes.

The interpreter is designed to execute bytecodes very efficiently. It can switch between running bytecodes and handing control to the platform-specific machine-code produced by the JIT compiler. The JIT compiler is described in "The JIT compiler" on page 57.

Platform port layer

The ability to reuse the code for the JVM for numerous operating systems and processor architectures is made possible by the platform port layer.

The platform port layer is an abstraction of the native platform functions that are required by the JVM. Other components of the JVM are written in terms of the platform-neutral platform port layer functions. Further porting of the JVM requires the provision of implementations of the platform port layer facilities.

Memory management

This description of the Garbage Collector and Allocator provides background information to help you diagnose problems with memory management.

Memory management is explained under these headings:

- "Overview of memory management"
- "Allocation" on page 26
- "Detailed description of global garbage collection" on page 29
- "Generational Concurrent Garbage Collector" on page 37
- "How to do heap sizing" on page 45
- "Interaction of the Garbage Collector with applications" on page 46
- "How to coexist with the Garbage Collector" on page 47
- "Frequently asked questions about the Garbage Collector" on page 50

For detailed information about diagnosing Garbage Collector problems, see "Garbage Collector diagnostic data" on page 333.

See also the reference information in "Garbage Collector command-line options" on page 453.

Overview of memory management

Memory management contains the Garbage Collector and the Allocator. It is responsible for allocating memory in addition to collecting garbage. Because the task of memory allocation is small, compared to that of garbage collection, the term "garbage collection" usually also means "memory management".

This section includes:

- A summary of some of the diagnostic techniques related to memory management.
- An understanding of the way that the Garbage Collector works, so that you can design applications accordingly.

The Garbage Collector allocates areas of storage in the heap. These areas of storage define Java objects. When allocated, an object continues to be *live* while a reference (pointer) to it exists somewhere in the JVM; therefore the object is *reachable*. When an object ceases to be referenced from the active state, it becomes *garbage* and can be reclaimed for reuse. When this reclamation occurs, the Garbage Collector must process a possible finalizer and also ensure that any internal JVM resources that are associated with the object are returned to the pool of such resources.

Object allocation

Object allocation is driven by requests by applications, class libraries, and the JVM for storage of Java objects, which can vary in size and require different handling.

Every allocation requires a *heap lock* to be acquired to prevent concurrent thread access. To optimize this allocation, particular areas of the heap are dedicated to a thread, known as the TLH (thread local heap), and that thread can allocate from its TLH without having to lock out other threads. This technique delivers the best possible allocation performance for small objects. Objects are allocated directly from a thread local heap. A new object is allocated from this cache without needing to grab the heap lock. All objects less than 512 bytes (768 bytes on 64-bit JVMs) are allocated from the cache. Larger objects are allocated from the cache if they can be contained in the existing cache. This cache is often referred to as the *thread local heap* or TLH.

Reachable objects

Reachable objects are found using frames on the thread stack, roots and references.

The active state of the JVM is made up of the set of stacks that represents the threads, the static fields that are inside Java classes, and the set of local and global JNI references. All functions that are called inside the JVM itself cause a frame to be created on the thread stack. This information is used to find the *roots*. A *root* is an object which has a reference to it from outside the heap. These roots are then used to find references to other objects. This process is repeated until all reachable objects are found.

Garbage collection

When the JVM cannot allocate an object from the heap because of lack of contiguous space, a memory allocation fault occurs, and the Garbage Collector is called.

The first task of the Garbage Collector is to collect all the garbage that is in the heap. This process starts when any thread calls the Garbage Collector either indirectly as a result of allocation failure, or directly by a specific call to System.gc(). The first step is to acquire exclusive control on the virtual machine to prevent any further Java operations. Garbage collection can then begin.

Heap sizing problems

If the operation of the heap, using the default settings, does not give the best results for your application, there are actions that you can take.

For the majority of applications, the default settings work well. The heap expands until it reaches a steady state, then remains in that state, which should give a heap

occupancy (the amount of live data on the heap at any given time) of 70%. At this level, the frequency and pause time of garbage collection should be acceptable.

For some applications, the default settings might not give the best results. Listed here are some problems that might occur, and some suggested actions that you can take. Use **verbose:gc** to help you monitor the heap.

The frequency of garbage collections is too high until the heap reaches a steady state.
Use **verbose:gc** to determine the size of the heap at a steady state and set **-Xms** to this value.

The heap is fully expanded and the occupancy level is greater than 70%.
Increase the **-Xmx** value so that the heap is not more than 70% occupied. The maximum heap size should, if possible, be able to be contained in physical memory to avoid paging. For the best performance, try to ensure that the heap never pages.

At 70% occupancy the frequency of garbage collections is too great.
Change the setting of **-Xminf**. The default is 0.3, which tries to maintain 30% free space by expanding the heap. A setting of 0.4, for example, increases this free space target to 40%, and reduces the frequency of garbage collections.

Pause times are too long.
If your application uses many short-lived objects, or is transaction-based (that is, objects in the transaction do not survive beyond the transaction commit), or if the heap space is fragmented, try using the **-Xgcpolicy:gencon** garbage collection policy. This policy treats short-lived objects differently from long-lived objects, and can reduce pause times and heap fragmentation.

In other situations, if a reduction in throughput is acceptable, try using the **-Xgcpolicy:optavgpause** policy. This policy reduces the pause times and makes them more consistent when the heap occupancy rises. It does, however, reduce throughput by approximately 5%, although this value varies with different applications.

If pause times are unacceptable during a global garbage collection, due to a large heap size, try using **-Xgcpolicy:balanced**. The balanced garbage collection policy can also address frequent class unloading issues, where many class loaders are being created, but require a global collection to unload. This policy is available for 64-bit platforms and must be used with the **-Xcompressedrefs** option. The policy is intended for environments where heap sizes are greater than 4 GB.

Here are some useful tips:

- Ensure that the heap never pages; that is, the maximum heap size must be able to be contained in physical memory.
- Avoid finalizers. You cannot guarantee when a finalizer will run, and often they cause problems. If you do use finalizers, try to avoid allocating objects in the finalizer method. A **verbose:gc** trace shows whether finalizers are being called.
- Avoid compaction. A **verbose:gc** trace shows whether compaction is occurring. Compaction is usually caused by requests for large memory allocations. Analyze requests for large memory allocations and avoid them if possible. If they are large arrays, for example, try to split them into smaller arrays.

Allocation

The Allocator is a component of memory management that is responsible for allocating areas of memory for the JVM. The task of memory allocation is small, compared to that of garbage collection.

Heap lock allocation

Heap lock allocation occurs when the allocation request cannot be satisfied in the existing cache.

As the name implies, heap lock allocation requires a lock and is therefore avoided, if possible, by using the cache. For a description of cache allocation, see "Cache allocation"

If the Garbage Collector cannot find a large enough chunk of free storage, allocation fails and the Garbage Collector must run a garbage collection. After a garbage collection cycle, if the Garbage Collector created enough free storage, it searches the freelist again and picks up a free chunk. The heap lock is released either after the object is allocated, or if not enough free space is found. If the Garbage Collector does not find enough free storage, it returns `OutOfMemoryError`.

Cache allocation

Cache allocation is specifically designed to deliver the best possible allocation performance for small objects.

Objects are allocated directly from a thread local allocation buffer that the thread has previously allocated from the heap. A new object is allocated from this cache without the need to grab the heap lock; therefore, cache allocation is very efficient.

All objects less than 512 bytes (768 bytes on 64-bit JVMs) are allocated from the cache. Larger objects are allocated from the cache if they can be contained in the existing cache; if not a locked heap allocation is performed.

The cache block is sometimes called a thread local heap (TLH). The size of the TLH varies from 512 bytes (768 on 64-bit JVMs) to 128 KB, depending on the allocation rate of the thread. Threads which allocate lots of objects are given larger TLHs to further reduce contention on the heap lock.

Large Object Area

The Large Object Areas (LOA) is an area of the tenure area of the heap set used solely to satisfy allocations for large objects. The LOA is used when the allocation request cannot be satisfied in the main area (also known as the small object area (SOA)) of the tenure heap.

As objects are allocated and freed, the heap can become fragmented in such a way that allocation can be met only by time-consuming compactions. This problem is more pronounced if an application allocates large objects. In an attempt to alleviate this problem, the large object area (LOA) is allocated. A large object in this context is considered to be any object 64 KB or greater in size. Allocations for new TLH objects are not considered to be large objects. The large object area is allocated by default for all GC polices except **-Xgcpolicy:balanced** but, if it is not used, it is shrunk to zero after a few collections. It can be disabled explicitly by specifying the **-Xnoloa** command-line option.

The Balanced Garbage Collection policy does not use the LOA. Therefore, when specifying **-Xgcpolicy:balanced**, any LOA options passed on the command line are ignored. The policy addresses the issues of LOA by reorganizing object layout with

the JVM to reduce heap fragmentation and compaction requirements. This change is contained completely within the JVM, and requires no knowledge or code changes in Java.

Initialization and the LOA:

The LOA boundary is calculated when the heap is initialized, and recalculated after every garbage collection. The size of the LOA can be controlled using command-line options: **-Xloainitial** and **-Xloamaximum**.

The options take values between 0 and 0.95 (0% thru 95% of the current tenure heap size). The defaults are:

- **-Xloainitial0.05** (5%)
- **-Xloaminimum0** (0%)
- **-Xloamaximum0.5** (50%)

Expansion and shrinkage of the LOA:

The Garbage Collector expands or shrinks the LOA, depending on usage.

The Garbage Collector uses the following algorithm:

- If an allocation failure occurs in the SOA:
  - If the current size of the LOA is greater than its initial size and if the amount of free space in the LOA is greater than 70%, reduce by 1% the percentage of space that is allocated to the LOA.
  - If the current size of the LOA is equal to or less than its initial size, and if the amount of free space in the LOA is greater than 90%:
    - If the current size of the LOA is greater than 1% of the heap, reduce by 1% the percentage of space that is allocated to the LOA.
    - If the current size of the LOA is 1% or less of the heap, reduce by 0.1%, the percentage of space that is allocated to the LOA.
- If an allocation failure occurs on the LOA:
  - If the size of the allocation request is greater than 20% of the current size of the LOA, increase the LOA by 1%.
  - If the current size of the LOA is less than its initial size, and if the amount of free space in the LOA is less than 50%, increase the LOA by 1%.
  - If the current size of the LOA is equal to or greater than its initial size, and if the amount of free space in the LOA is less than 30%, increase the LOA by 1%.

Allocation in the LOA:

The size of the request determines where the object is allocated.

When allocating an object, the allocation is first attempted in the Small Object Area (SOA). If it is not possible to find a free entry of sufficient size to satisfy the allocation, and the size of the request is equal to or greater than 64 KB, the allocation is tried in the LOA again. If the size of the request is less than 64 KB or insufficient contiguous space exists in the LOA, an allocation failure is triggered.

Compressed references

When using compressed references, the JVM stores all references to objects, classes, threads, and monitors as 32-bit values. Use the **-Xcompressedrefs** and

-Xnocompressedrefs command-line options to enable or disable compressed references in a 64-bit JVM. Only 64-bit JVMs recognize these options.

Note: If you are using compressed references on z/OS v1.10 or earlier, you must use APAR OA26294.

The use of compressed references improves the performance of many applications because objects are smaller, resulting in less frequent garbage collection and improved memory cache utilization. Certain applications might not benefit from compressed references. Test the performance of your application with and without compressed references to determine if they are appropriate. For default option settings, see "JVM command-line options" on page 428.

Using compressed references runs a different version of the JVM. You need to enable compressed references when using the dump extractor to analyze dumps produced by the JVM, see "Using the dump viewer" on page 273.

When you are using compressed references, the following structures are allocated in the lowest 4 GB of the address space:

- Classes
- Threads
- Monitors

Additionally, the operating system and native libraries use some of this address space. Small Java heaps are also allocated in the lowest 4 GB of the address space. Larger Java heaps are allocated higher in the address space.

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. You can often resolve these errors with a larger **-Xmx** option to put the Java heap higher in the address space.

A command-line option can be used with `-Xcompressedrefs` to allocate the heap you specify with the `-Xmx` option, in a memory range of your choice. This option is `-Xgc:preferredHeapBase=<address>`, where *<address>* is the base memory address for the heap. In the following example, the heap is located at the 4GB mark, leaving the lowest 4GB of address space for use by other processes.

```
-Xgc:preferredHeapBase=0x100000000
```

If the heap cannot be allocated in a contiguous block at the `preferredHeapBase` address you specified, an error occurs detailing a Garbage Collection (GC) allocation failure startup. When the **-Xgc:preferredHeapBase** option is used with the **-Xlp** option, the `preferredHeapBase` address must be a multiple of the large page size. If you specify an inaccurate heap base address, the heap is allocated with the default page size.

64-bit JVMs recognize the following Oracle JVM options:

-XX:+UseCompressedOops
: This enables compressed references in 64-bit JVMs. It is identical to specifying the `-Xcompressedrefs` option.

-XX:-UseCompressedOops
: This prevents use of compressed references in 64-bit JVMs.

Note: These options are provided to help when porting applications from the Oracle JVM to the IBM JVM, for 64-bit platforms. The options might not be supported in subsequent releases.

Detailed description of global garbage collection

Garbage collection is performed when an allocation failure occurs in heap lock allocation, or if a specific call to System.gc() occurs. The thread that has the allocation failure or the System.gc() call takes control and performs the garbage collection.

The first step in garbage collection is to acquire exclusive control on the Virtual machine to prevent any further Java operations. Garbage collection then goes through the three phases: mark, sweep, and, if required, compaction. The IBM Garbage Collector (GC) is a stop-the-world (STW) operation, because all application threads are stopped while the garbage is collected.

A global garbage collection occurs only in exceptional circumstances when using the Balanced Garbage Collection policy. Circumstances that might cause this rare event include:

- A System.gc() call.
- A request by tooling.
- A combination of heap size, occupied heap memory, and collection rates that cannot keep up with demand.

Mark phase

In mark phase, all the live objects are marked. Because unreachable objects cannot be identified singly, all the reachable objects must be identified. Therefore, everything else must be garbage. The process of marking all reachable objects is also known as tracing.

The mark phase uses:

- A pool of structures called *work packets*. Each work packet contains a mark stack. A mark stack contains references to live objects that have not yet been traced. Each marking thread refers to two work packets;
  1. An input packet from which references are popped.
  2. An output packet to which unmarked objects that have just been discovered are pushed.

  References are marked when they are pushed onto the output packet. When the input packet becomes empty, it is added to a list of empty packets and replaced by a non-empty packet. When the output packet becomes full it is added to a list of non-empty packets and replaced by a packet from the empty list.
- A bit vector called the *mark bit array* identifies the objects that are reachable and have been visited. This bit array, also known as the *mark map*, is allocated by the JVM at startup based on the maximum heap size **(-Xmx)**.The mark bit array contains one bit for each 8 bytes of heap space. The bit that corresponds to the start address for each reachable object is set when it is first visited.

The first stage of tracing is the identification of root objects. The active state of the JVM consists of:

- The saved registers for each thread
- The set of stacks that represent the threads
- The static fields that are in Java classes

- The set of local and global JNI references.

All functions that are called in the JVM itself cause a frame on the C stack. This frame might contain references to objects as a result of either an assignment to a local variable, or a parameter that is sent from the caller. All these references are treated equally by the tracing routines.

All the mark bits for all root objects are set and references to the roots pushed to the output work packet. Tracing then proceeds by iteratively popping a reference off the marking thread's input work packet and then scanning the referenced object for references to other objects. If the mark bit is off, there are references to unmarked objects. The object is marked by setting the appropriate bit in the mark bit array. The reference is then pushed to the output work packet of the marking thread. This process continues until all the work packets are on the empty list, at which point all the reachable objects have been identified.

Mark stack overflow:

Because the set of work packets has a finite size, it can overflow and the Garbage Collector (GC) then performs a series of actions.

If an overflow occurs, the GC empties one of the work packets by popping its references one at a time, and chaining the referenced objects off their owning class by using the class pointer field in the object header. All classes with overflow objects are also chained together. Tracing can then continue as before. If a further mark stack overflow occurs, more packets are emptied in the same way.

When a marking thread asks for a new non-empty packet and all work packets are empty, the GC checks the list of overflow classes. If the list is not empty, the GC traverses this list and repopulates a work packet with the references to the objects on the overflow lists. These packets are then processed as described previously. Tracing is complete when all the work packets are empty and the overflow list is empty.

Parallel mark:

The goal of parallel mark is to increase typical mark performance on a multiprocessor system, while not degrading mark performance on a uniprocessor system.

The performance of object marking is increased through the addition of helper threads that share the use of the pool of work packets. For example, full output packets that are returned to the pool by one thread can be picked up as new input packets by another thread.

Parallel mark still requires the participation of one application thread that is used as the master coordinating agent. The helper threads assist both in the identification of the root pointers for the collection and in the tracing of these roots. Mark bits are updated by using host machine atomic primitives that require no additional lock.

For information about the number of helper threads that are created, and how you can change that number, see “Frequently asked questions about the Garbage Collector” on page 50.

Concurrent mark:

Concurrent mark gives reduced and consistent garbage collection pause times when heap sizes increase.

The GC starts a concurrent marking phase before the heap is full. In the concurrent phase, the GC scans the heap, inspecting "root" objects such as stacks, JNI references, and class static fields. The stacks are scanned by asking each thread to scan its own stack. These roots are then used to trace live objects concurrently. Tracing is done by a low-priority background thread and by each application thread when it does a heap lock allocation.

While the GC is marking live objects concurrently with application threads running, it must record any changes to objects that are already traced. It uses a write barrier that is run every time a reference in an object is updated. The write barrier flags when an object reference update has occurred. The flag is used to force a rescan of part of the heap.

The heap is divided into 512 byte sections. Each section is allocated a single-byte card in the card table. Whenever a reference to an object is updated, the card that corresponds to the start address of the object that has been updated with the new object reference is marked with the hex value `0x01`. A byte is used instead of a bit to eliminate contention, by allowing cards to be marked using non-atomic operations. A stop-the-world (STW) collection is started when one of the following events takes place:

- An allocation failure occurs.
- A System.gc call is made.
- Concurrent mark finishes all the possible marking.

The GC tries to start the concurrent mark phase so that it finishes at the same time as the heap is exhausted. The GC identifies the optimum start time by constant tuning of the parameters that govern the concurrent mark time. In the STW phase, the GC rescans all roots, then uses the marked cards to see what else must be retraced. The GC then sweeps as normal. It is guaranteed that all objects that were unreachable at the start of the concurrent phase are collected. It is not guaranteed that objects that become unreachable during the concurrent phase are collected. Objects which become unreachable during the concurrent phase are known as "floating garbage".

Reduced and consistent pause times are the benefits of concurrent mark, but they come at a cost. Application threads must do some tracing when they are requesting a heap lock allocation. The processor usage needed varies depending on how much idle processor time is available for the background thread. Also, the write barrier requires additional processor usage.

The **-Xgcpolicy** command-line parameter is used to enable and disable concurrent mark:

-Xgcpolicy: <gencon | optavgpause | optthruput | subpool | balanced>

The **-Xgcpolicy** options have these effects:

gencon Enables concurrent mark, and uses it in combination with generational garbage collection to help minimize the time that is spent in any garbage collection pause. **gencon** is the default setting. If you are having problems with erratic application response times that are caused by normal garbage

collections, you can reduce those problems, reduce heap fragmentation, and still maintain good throughput, by using the **gencon** option. This option is particularly useful for applications that use many short-lived objects.

optavgpause
Enables concurrent mark with its default values. If you are having problems with erratic application response times that are caused by normal garbage collections, you can reduce those problems at the cost of some throughput, by using the **optavgpause** option.

optthruput
Disables concurrent mark. If you do not have pause time problems (as seen by erratic application response times), you get the best throughput with this option.

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**.

balanced
Disables concurrent mark. This policy does use concurrent garbage collection technology, but not in the way that concurrent mark is implemented here. For more information, see "Global Mark Phase" on page 42.

Sweep phase

On completion of the mark phase the mark bit vector identifies the location of all the live objects in the heap. The sweep phase uses this to identify those chunks of heap storage that can be reclaimed for future allocations; these chunks are added to the pool of free space.

A free chunk is identified by examining the mark bit vector looking for sequences of zeros, which identify possible free space. GC ignores any sequences of zeros that correspond to a length less than the minimum free size. When a sequence of sufficient length is found, the GC checks the length of the object at the start of the sequence to determine the actual amount of free space that can be reclaimed. If this amount is greater than or equal to the minimum size for a free chunk, it is reclaimed and added to the free space pool. The minimum size for a free chunk is currently defined as 512 bytes on 32-bit platforms, and 768 bytes on 64-bit platforms.

The small areas of storage that are not on the freelist are known as "dark matter", and they are recovered when the objects that are next to them become free, or when the heap is compacted. It is not necessary to free the individual objects in the free chunk, because it is known that the whole chunk is free storage. When a chunk is freed, the GC has no knowledge of the objects that were in it.

Parallel bitwise sweep:

Parallel bitwise sweep improves the sweep time by using available processors. In parallel bitwise sweep, the Garbage Collector uses the same helper threads that are used in parallel mark, so the default number of helper threads is also the same and can be changed with the **-Xgcthreads** option.

The heap is divided into sections of 256 KB and each thread (helper or master) takes a section at a time and scans it, performing a modified bitwise sweep. The results of this scan are stored for each section. When all sections have been scanned, the freelist is built.

With the Balanced Garbage Collection policy, **-Xgcpolicy:balanced**, the Java heap is divided into approximately 1000 sections, providing a granular base for the parallel bitwise sweep.

Concurrent sweep:

Like concurrent mark, concurrent sweep gives reduced garbage collection pause times when heap sizes increase. Concurrent sweep starts immediately after a stop-the-world (STW) collection, and must at least finish a certain subset of its work before concurrent mark is allowed to kick off, because the mark map used for concurrent mark is also used for sweeping.

The concurrent sweep process is split into two types of operations:

- **Sweep analysis:** Sections of data in the mark map (mark bit array) are analyzed for ranges of free or potentially free memory.
- **Connection:** The analyzed sections of the heap are connected into the free list.

Heap sections are calculated in the same way as for parallel bitwise sweep.

An STW collection initially performs a minimal sweep operation that searches for and finds a free entry large enough to satisfy the current allocation failure. The remaining unprocessed portion of the heap and mark map are left to concurrent sweep to be both analyzed and connected. This work is accomplished by Java threads through the allocation process. For a successful allocation, an amount of heap relative to the size of the allocation is analyzed, and is performed outside the allocation lock. In an allocation, if the current free list cannot satisfy the request, sections of analyzed heap are found and connected into the free list. If sections exist but are not analyzed, the allocating thread must also analyze them before connecting.

Because the sweep is incomplete at the end of the STW collection, the amount of free memory reported (through verbose garbage collection or the API) is an estimate based on past heap occupancy and the ratio of unprocessed heap size against total heap size. In addition, the mechanics of compaction require that a sweep is completed before a compaction can occur. Consequently, an STW collection that compacts does not have concurrent sweep active during the next round of execution.

To enable concurrent sweep, use the **-Xgcpolicy:** parameter **optavgpause**. It becomes active along with concurrent mark. The modes **optthruput**, **balanced**, and **gencon** do not support concurrent sweep.

Compaction phase

When the garbage has been removed from the heap, the Garbage Collector can consider compacting the resulting set of objects to remove the spaces that are between them. The process of compaction is complicated because, if any object is moved, the GC must change all the references that exist to it. The default is not to compact.

The following analogy might help you understand the compaction process. Think of the heap as a warehouse that is partly full of pieces of furniture of different

sizes. The free space is the gaps between the furniture. The free list contains only gaps that are larger than a particular size. Compaction pushes everything in one direction and closes all the gaps. It starts with the object that is closest to the wall, and puts that object against the wall. Then it takes the second object in line and puts that against the first. Then it takes the third and puts it against the second, and so on. At the end, all the furniture is at one end of the warehouse and all the free space is at the other.

To keep compaction times to a minimum, the helper threads are used again.

Compaction occurs if any one of the following conditions are true and **-Xnocompactgc** has not been specified:

- **-Xcompactgc** has been specified.
- Following the sweep phase, not enough free space is available to satisfy the allocation request.
- A System.gc() has been requested and the last allocation failure triggering a global garbage collection did not compact or **-Xcompactexplicitgc** has been specified.
- At least half the previously available memory has been consumed by TLH allocations (ensuring an accurate sample) and the average TLH size falls to less than 1024 bytes
- The scavenger is enabled, and the largest object that the scavenger failed to tenure in the most recent scavenge is larger than the largest free entry in tenured space.
- The heap is fully expanded and less than 4% of old space is free.
- Less than 128 KB of the heap is free.

With the Balanced Garbage Collection policy, the **-Xcompactgc** and **-Xnocompactgc** options are respected only if a global garbage collection is required. A global garbage collection occurs in rare circumstances, as described in "Detailed description of global garbage collection" on page 29. All other collection activity for the Balanced policy is subject to possible compaction or object movement.

Reference objects

When a reference object is created, it is added to a list of reference objects of the same type. The referent is the object to which the reference object points.

Instances of SoftReference, WeakReference, and PhantomReference are created by the user and cannot be changed; they cannot be made to refer to objects other than the object that they referenced on creation.

If an object has a class that defines a finalize method, a pointer to that object is added to a list of objects that require finalization.

During garbage collection, immediately following the mark phase, these lists are processed in a specific order:

1. Soft
2. Weak
3. Final
4. Phantom

Soft, weak, and phantom reference processing:

The Garbage Collector (GC) determines if a reference object is a candidate for collection and, if so, performs a collection process that differs for each reference type. Soft references are collected if their referent is not marked and if #get() has not been called on the reference object for a number of garbage collection cycles. Weak and phantom references are always collected if their referent is not marked.

For each element on a list, GC determines if the reference object is eligible for processing and then if it is eligible for collection.

An element is eligible for processing if it is marked and has a non-null referent field. If this is not the case, the reference object is removed from the reference list, resulting in it being freed during the sweep phase.

If an element is determined to be eligible for processing, GC must determine if it is eligible for collection. The first criterion here is simple. Is the referent marked? If it is marked, the reference object is not eligible for collection and GC moves onto the next element of the list.

If the referent is not marked, GC has a candidate for collection. At this point the process differs for each reference type. Soft references are collected if their referent has not been marked for a number of garbage collection cycles. The number of garbage collection cycles depends on the percentage of free heap space. You adjust the frequency of collection with the **-Xsoftrefthreshold** option. For more information about using **-Xsoftrefthreshold**, see "Garbage Collector command-line options" on page 453. If there is a shortage of available storage, all soft references are cleared. All soft references are guaranteed to have been cleared before the OutOfMemoryError is thrown.

Weak and phantom references are always collected if their referent is not marked. When a phantom reference is processed, its referent is marked so it will persist until the following garbage collection cycle or until the phantom reference is processed if it is associated with a reference queue. When it is determined that a reference is eligible for collection, it is either queued to its associated reference queue or removed from the reference list.

Final reference processing

The processing of objects that require finalization is more straightforward.

1. The list of objects is processed. Any element that is not marked is processed by:
   a. Marking and tracing the object
   b. Creating an entry on the finalizable object list for the object
2. The GC removes the element from the unfinalized object list.
3. The final method for the object is run at an undetermined point in the future by the reference handler thread.

JNI weak reference

JNI weak references provide the same capability as that of WeakReference objects, but the processing is very different. A JNI routine can create a JNI Weak reference to an object and later delete that reference. The Garbage Collector clears any weak reference where the referent is unmarked, but no equivalent of the queuing mechanism exists.

Failure to delete a JNI Weak reference causes a memory leak in the table and performance problems. This also applies to JNI global references. The processing of

JNI weak references is handled last in the reference handling process. The result is that a JNI weak reference can exist for an object that has already been finalized and had a phantom reference queued and processed.

Heap expansion

Heap expansion occurs after garbage collection while exclusive access of the virtual machine is still held. The heap is expanded in a set of specific situations.

The active part of the heap is expanded up to the maximum if one of three conditions is true:

- The Garbage Collector (GC) did not free enough storage to satisfy the allocation request.
- Free space is less than the minimum free space, which you can set by using the **-Xminf** parameter. The default is 30%.
- More than the maximum time threshold is being spent in garbage collection, set using the **-Xmaxt** parameter. The default is 13%.

The amount to expand the heap is calculated as follows:

1. The **-Xminf** option specifies the minimum percentage of heap to remain free after a garbage collection. If the heap is being expanded to satisfy this value, the GC calculates how much heap expansion is required.

   You can set the maximum expansion amount using the **-Xmaxe** parameter. The default value is 0, which means there is no maximum expansion limit. If the calculated required heap expansion is greater than the non-zero value of **-Xmaxe**, the required heap expansion is reduced to the value of **-Xmaxe**.

   You can set the minimum expansion amount using the **-Xmine** parameter. The default value is 1 MB. If the calculated required heap expansion is less than the value of **-Xmine**, the required heap expansion is increased to the value of **-Xmine**.
2. If the heap is expanding and the JVM is spending more than the maximum time threshold, the GC calculates how much heap expansion is needed to provide 17% free space. The expansion is adjusted as described in the previous step, depending on **-Xmaxe** and **-Xmine**.
3. If garbage collection did not free enough storage, the GC ensures that the heap is expanded by at least the value of the allocation request.

All calculated expansion amounts are rounded to the nearest 512-byte boundary on 32-bit JVMs or a 1024-byte boundary on 64-bit JVMs.

Heap shrinkage

Heap shrinkage occurs after garbage collection while exclusive access of the virtual machine is still held. Shrinkage does not occur in a set of specific situations. Also, there is a situation where a compaction occurs before the shrink.

Shrinkage does not occur if any of the following conditions are true:

- The Garbage Collector (GC) did not free enough space to satisfy the allocation request.
- The maximum free space, which can be set by the **-Xmaxf** parameter (default is 60%), is set to 100%.
- The heap has been expanded in the last three garbage collections.
- This is a System.gc() and the amount of free space at the beginning of the garbage collection was less than **-Xminf** (default is 30%) of the live part of the heap.

- If none of the previous options are true, and more than **-Xmaxf** free space exists, the GC must calculate how much to shrink the heap to get it to **-Xmaxf** free space, without dropping to less than the initial (**-Xms**) value. This figure is rounded down to a 512-byte boundary on 32-bit JVMs or a 1024-byte boundary on 64-bit JVMs.

A compaction occurs before the shrink if all the following conditions are true:

- A compaction was not done on this garbage collection cycle.
- No free chunk is at the end of the heap, or the size of the free chunk that is at the end of the heap is less than 10% of the required shrinkage amount.
- The GC did not shrink and compact on the last garbage collection cycle.

On initialization, the JVM allocates the whole heap in a single contiguous area of virtual storage. The amount that is allocated is determined by the setting of the **-Xmx** parameter. No virtual space from the heap is ever freed back to the native operating system. When the heap shrinks, it shrinks inside the original virtual space.

Whether any physical memory is released depends on the ability of the native operating system. If it supports *paging*; the ability of the native operating system to commit and decommit physical storage to the virtual storage; the GC uses this function. In this case, physical memory can be decommitted on a heap shrinkage.

You never see the amount of virtual storage that is used by the JVM decrease. You might see physical memory free size increase after a heap shrinkage. The native operating system determines what it does with decommitted pages.

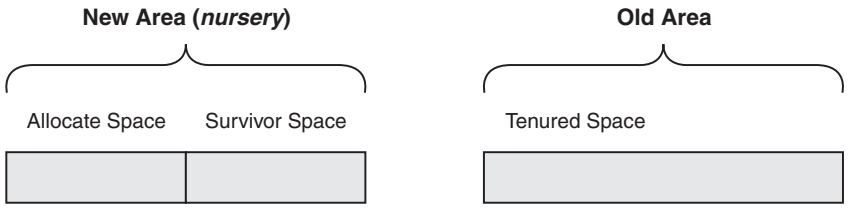
Where paging is supported, the GC allocates physical memory to the initial heap to the amount that is specified by the **-Xms** parameter. Additional memory is committed as the heap grows.

Generational Concurrent Garbage Collector

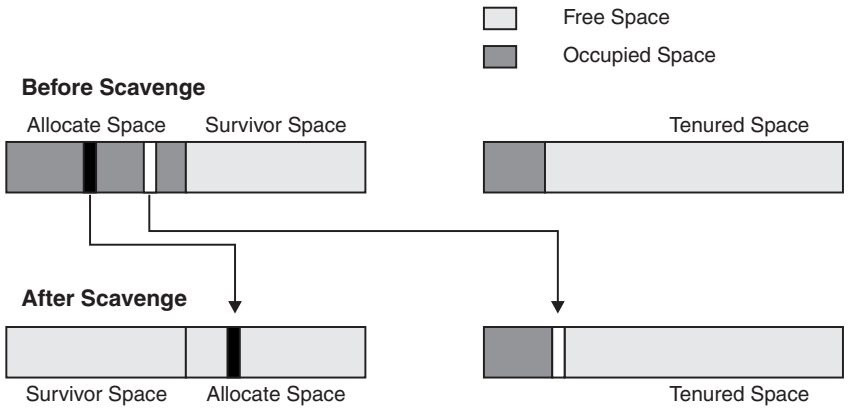
The Generational Concurrent Garbage Collector has been introduced in Java 5.0 from IBM. A generational garbage collection strategy is well suited to an application that creates many short-lived objects, as is typical of many transactional applications.

You activate the Generational Concurrent Garbage Collector with the **-Xgcpolicy:gencon** command-line option.

The Java heap is split into two areas, a new (or nursery) area and an old (or tenured) area. Objects are created in the new area and, if they continue to be reachable for long enough, they are moved into the old area. Objects are moved when they have been reachable for enough garbage collections (known as the tenure age).



The new area is split into two logical spaces: allocate and survivor. Objects are allocated into the Allocate Space. When that space is filled, a garbage collection process called scavenge is triggered. During a scavenge, reachable objects are copied either into the Survivor Space or into the Tenured Space if they have reached the tenured age. Objects in the new area that are not reachable remain untouched. When all the reachable objects have been copied, the spaces in the new area switch roles. The new Survivor Space is now entirely empty of reachable objects and is available for the next scavenge.



This diagram illustrates what happens during a scavenge. When the Allocate Space is full, a garbage collection is triggered. Reachable objects are then traced and copied into the Survivor Space. Objects that have reached the tenure age (have already been copied inside the new area a number of times) are promoted into Tenured Space. As the name Generational Concurrent implies, the policy has a concurrent aspect to it. The Tenured Space is concurrently traced with a similar approach to the one used for **-Xgcpolicy:optavgpause**. With this approach, the pause time incurred from Tenured Space collections is reduced.

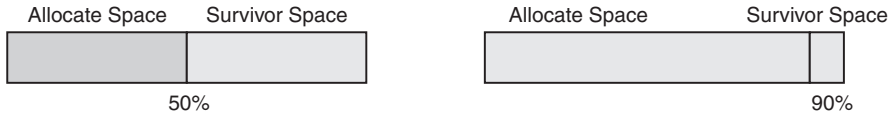
Tenure age

Tenure age is a measure of the object age at which it should be promoted to the tenure area. This age is dynamically adjusted by the JVM and reaches a maximum value of 14. An object's age is incremented on each scavenge. A tenure age of x means that an object is promoted to the tenure area after it has survived x flips between survivor and allocate space. The threshold is adaptive and adjusts the tenure age based on the percentage of space used in the new area.

Tilt ratio

The size of the allocate space in the new area is maximized by a technique called tilting. Tilting controls the relative sizes of the allocate and survivor spaces. Based on the amount of data that survives the scavenge, the ratio is adjusted to maximize the amount of time between scavenges.

For example, if the initial total new area size is 500 MB, the allocate and survivor spaces start with 250 MB each (a 50% split). As the application runs and a scavenge GC event is triggered, only 50 MB survives. In this situation, the survivor space is decreased, allowing more space for the allocate space. A larger allocate area means that it takes longer for a garbage collection to occur. This diagram illustrates how the boundary between allocate and survivor space is affected by the tilt ratio.



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.

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, which allows the collector to focus only on the regions which offer the best return on investment.

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.

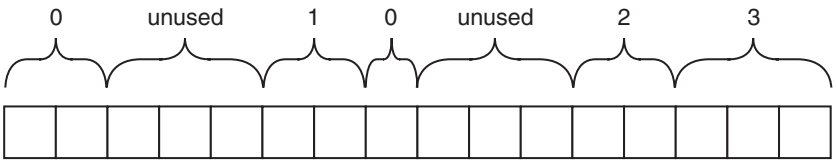
Region age

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.

Partial Garbage Collection

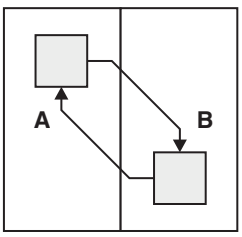
A Partial Garbage Collection (PGC) reclaims memory by using either a Copy-Forward or Mark-Compact operation on the Java heap.

Note: The **-Xpartialcompactgc** option, which in previous version of IBM Java enabled partial compaction, is now deprecated and has no effect if used.

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:



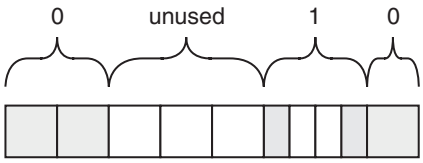
If only A or B is collected, one half of the cycle keeps the other alive. However, a GMP can see that these objects are unreachable.

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.

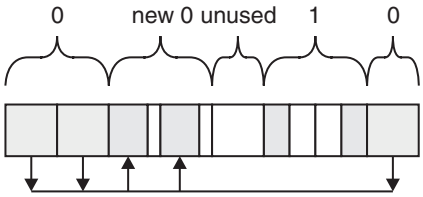
Copy-Forward operation

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

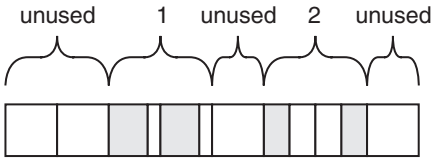
This diagram shows the Java heap before the Copy-Forward operation:



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



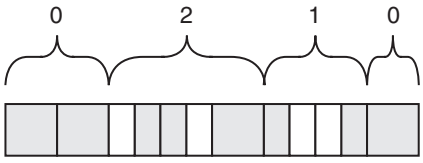
This diagram shows the Java heap after the Copy-Forward operation, where region ages have been incremented:



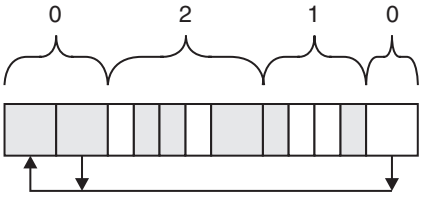
Mark-Compact operation

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

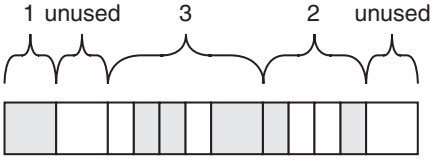
This diagram shows the Java heap before the Mark-Compact operation:



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

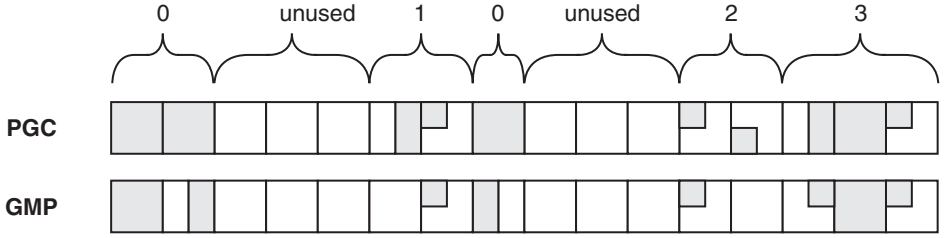
A Global Mark Phase (GMP) takes place on the entire Java heap, finding, and marking abandoned objects for garbage collection.

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.

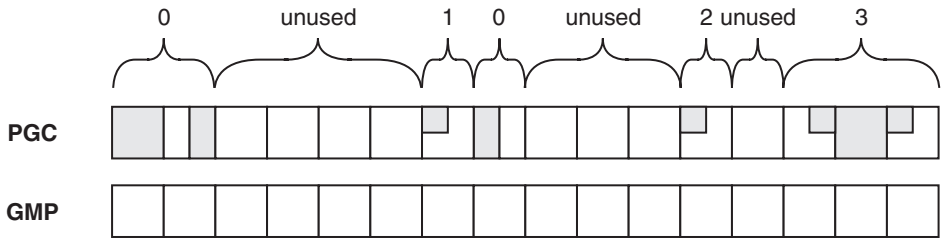
GMP also performs some work concurrently with the application threads. This concurrent mark operation is based purely on background threads, which allows idle processors to complete work, no matter how quickly the application is allocating memory. This concurrent mark operation is unlike the concurrent mark operations that are specified with **-Xgcpolicy:gencon** or **-Xgcpolicy:optavgpause**. For more information about the use of concurrent mark with these options, see "Concurrent mark" on page 31.

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 for Java - Garbage Collection and Memory Visualizer and IBM Monitoring and Diagnostic Tools for Java - 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 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.

How to do heap sizing

You can do heap sizing to suit your requirements.

Generally:

- Do not start with a minimum heap size that is the same as the maximum heap size.
- Use **-verbose:gc** to tailor the minimum and maximum settings.
- Investigate the use of fine-tuning options.

Initial and maximum heap sizes

Understanding the operations of the Garbage Collector (GC) helps you set initial and maximum heap sizes for efficient management of the heap.

When you have established the maximum heap size that you need, you might want to set the minimum heap size to the same value; for example, **-Xms512M -Xmx512M**. However, using the same values is typically not a good idea, because it delays the start of garbage collection until the heap is full. Therefore, the first time that the GC runs, the process can take longer. Also, the heap is more likely to be fragmented and require a heap compaction. You are advised to start your application with the minimum heap size that your application requires. When the GC starts up, it will run frequently and efficiently, because the heap is small.

If the GC cannot find enough garbage, it runs compaction. If the GC finds enough garbage, or any of the other conditions for heap expansion are met (see “Heap expansion” on page 36), the GC expands the heap.

Therefore, an application typically runs until the heap is full. Then, successive garbage collection cycles recover garbage. When the heap is full of live objects, the GC compacts the heap. If sufficient garbage is still not recovered, the GC expands the heap.

From the earlier description, you can see that the GC compacts the heap as the needs of the application rise, so that as the heap expands, it expands with a set of compacted objects in the bottom of the original heap. This process is an efficient way to manage the heap, because compaction runs on the smallest-possible heap size at the time that compaction is found to be necessary. Compaction is performed with the minimum heap sizes as the heap grows. Some evidence exists that an application's initial set of objects tends to be the key or root set, so that compacting them early frees the remainder of the heap for more short-lived objects.

Eventually, the JVM has the heap at maximum size with all long-lived objects compacted at the bottom of the heap. The compaction occurred when compaction was in its least expensive phase. The amount of processing and memory usage required to expand the heap is almost trivial compared to the cost of collecting and compacting a very large fragmented heap.

Using verbose:gc

You can use **-verbose:gc** when running your application with no load, and again under stress, to help you set the initial and maximum heap sizes.

The **-verbose:gc** output is fully described in "Garbage Collector diagnostic data" on page 333. Turn on **-verbose:gc** and run up the application with no load. Check the heap size at this stage. This provides a rough guide to the start size of the heap (**-Xms** option) that is needed. If this value is much larger than the defaults (see "Default settings for the JVM" on page 474), think about reducing this value a little to get efficient and rapid compaction up to this value, as described in "Initial and maximum heap sizes" on page 45.

By running an application under stress, you can determine a maximum heap size. Use this to set your max heap (**-Xmx**) value.

Using fine tuning options

You can change the minimum and maximum vales of the free space after garbage collection, the expansion amount, and the garbage collection time threshold, to fine tune the management of the heap.

Consider the description of the following command-line parameters and consider applying them to fine tune the way the heap is managed:

-Xminf and -Xmaxf
: Minimum and maximum free space after garbage collection.

-Xmine and -Xmaxe
: Minimum and maximum expansion amount.

-Xmint and -Xmaxt
: Minimum and maximum garbage collection time threshold.

These are also described in "Heap expansion" on page 36 and "Heap shrinkage" on page 36.

Interaction of the Garbage Collector with applications

Understanding the way the Garbage Collector works helps you to understand its relationship with your applications.

The Garbage Collector behaves in these ways:

1. The Garbage Collector will collect some (but not necessarily all) unreachable objects.
2. The Garbage Collector will not collect reachable objects
3. The Garbage Collector will stop all threads when it is running.
4. The Garbage Collector will start in these ways:
   a. The Garbage Collector is triggered when an allocation failure occurs, but will otherwise not run itself.
   b. The Garbage Collector will accept manual calls unless the **-Xdisableexplicitgc** parameter is specified. A manual call to the Garbage Collector (for example, through the System.gc() call) suggests that a garbage collection cycle will run. In fact, the call is interpreted as a request for full garbage collection scan unless a garbage collection cycle is already running or explicit garbage collection is disabled by specifying **-Xdisableexplicitgc**.
5. The Garbage Collector will collect garbage at its own sequence and timing, subject to item 4b.

6. The Garbage Collector accepts all command-line variables and environment variables.
7. Note these points about finalizers:
   a. They are not run in any particular sequence.
   b. They are not run at any particular time.
   c. They are not guaranteed to run at all.
   d. They will run asynchronously to the Garbage Collector.

How to coexist with the Garbage Collector

Use this background information to help you diagnose problems in the coexistence of your applications with the Garbage Collector (GC).

Do not try to control the GC or to predict what will happen in a given garbage collection cycle. This unpredictability is handled, and the GC is designed to run well and efficiently inside these conditions.

Set up the initial conditions that you want and let the GC run. It will behave as described in "Interaction of the Garbage Collector with applications" on page 46, which is in the JVM specification.

Root set

The root set is an internally derived set of references to the contents of the stacks and registers of the JVM threads and other internal data structures at the time that the Garbage Collector was called.

This composition of the root set means that the graph of reachable objects that the Garbage Collector constructs in any given cycle is nearly always different from that traced in another cycle (see list item 5 in "Interaction of the Garbage Collector with applications" on page 46). This difference has significant consequences for finalizers (list item 7), which are described more fully in "Finalizers" on page 48.

Thread local heap

The Garbage Collector (GC) maintains areas of the heap for fast object allocation.

The heap is subject to concurrent access by all the threads that are running in the JVM. Therefore, it must be protected by a resource lock so that one thread can complete updates to the heap before another thread is allowed in. Access to the heap is therefore single-threaded. However, the GC also maintains areas of the heap as thread caches or thread local heap (TLH). These TLHs are areas of the heap that are allocated as a single large object, marked non-collectable, and allocated to a thread. The thread can now sub allocate from the TLH objects that are smaller than a defined size. No heap lock is needed which means that allocation is very fast and efficient. When a cache becomes full, a thread returns the TLH to the main heap and grabs another chunk for a new cache.

A TLH is not subject to a garbage collection cycle; it is a reference that is dedicated to a thread.

Bug reports

Attempts to predict the behavior of the Garbage Collector (GC) are frequent underlying causes of bug reports.

Here is an example of a regular bug report to Java service of the "Hello World" variety. A simple program allocates an object or objects, clears references to these

objects, and then initiates a garbage collection cycle. The objects are not seen as collected. Typically, the objects are not collected because the application has attached a finalizer that does not run immediately.

It is clear from the way that the GC works that more than one valid reason exists for the objects not being seen as collected:

- An object reference exists in the thread stack or registers, and the objects are retained garbage.
- The GC has not chosen to run a finalizer cycle at this time.

See list item 1 in "Interaction of the Garbage Collector with applications" on page 46. Real garbage is always found eventually, but it is not possible to predict when as stated in list item 5.

Finalizers

The Java service team recommends that applications avoid the use of finalizers if possible. The JVM specification states that finalizers are for emergency clear-up of, for example, hardware resources. The service team recommends that you use finalizers for this purpose only. Do not use them to clean up Java software resources or for closedown processing of transactions.

The reasons for this recommendation are partly because of the nature of finalizers and the permanent linkage to garbage collection, and partly because of the way garbage collection works as described in "Interaction of the Garbage Collector with applications" on page 46.

Nature of finalizers:

The JVM specification does not describe finalizers, except to state that they are final in nature. It does not state when, how, or whether a finalizer is run. Final, in terms of a finalizer, means that the object is known not to be in use any more.

The object is definitely not in use only when it is not reachable. Only the Garbage Collector (GC) can determine that an object is not reachable. Therefore, when the GC runs, it determines which are the unreachable objects that have a finalizer method attached. Normally, such objects are collected, and the GC can satisfy the memory allocation fault. Finalized garbage must have its finalizer run before it can be collected, so no finalized garbage can be collected in the cycle that finds it. Therefore, finalizers make a garbage collection cycle longer (the cycle has to detect and process the objects) and less productive. Finalizers use more of the processor and resources in addition to regular garbage collection. Because garbage collection is a stop-the-world operation, it is sensible to reduce the processor and resource usage as much as possible.

The GC cannot run finalizers itself when it finds them, because a finalizer might run an operation that takes a long time. The GC cannot risk locking out the application while this operation is running. Therefore, finalizers must be collected into a separate thread for processing. This task adds more processor usage into the garbage collection cycle.

Finalizers and garbage collection:

The behavior of the Garbage Collector (GC) affects the interaction between the GC and finalizers.

The way finalizers work, described in list item 7 in "Interaction of the Garbage Collector with applications" on page 46, indicates the non-predictable behavior of the GC. The significant results are:

- The graph of objects that the GC finds cannot be reliably predicted by your application. Therefore, the sequence in which finalized objects are located has no relationship to either
  - the sequence in which the finalized objects are created
  - the sequence in which the finalized objects become garbage.

  The sequence in which finalizers are run cannot be predicted by your application.
- The GC does not know what is in a finalizer, or how many finalizers exist. Therefore, the GC tries to satisfy an allocation without processing finalizers. If a garbage collection cycle cannot produce enough normal garbage, it might decide to process finalized objects. Therefore, it is not possible to predict when a finalizer is run.
- Because a finalized object might be garbage that is retained, a finalizer might not run at all.

How finalizers are run:

When the Garbage Collector (GC) decides to process unreachable finalized objects, those objects are placed onto a queue that is used as input to a separate finalizer thread.

When the GC has ended and the threads are unblocked, this finalizer thread starts. It runs as a high-priority thread and runs down the queue, running the finalizer of each object in turn. When the finalizer has run, the finalizer thread marks the object as collectable and the object is probably collected in the next garbage collection cycle. See list item 7d in "Interaction of the Garbage Collector with applications" on page 46. If you are running with a large heap, the next garbage collection cycle might not happen for some time.

Summary and alternative approach:

When you understand the characteristics and use of finalizers, consider an alternative approach to tidying Java resources.

Finalizers are an expensive use of computer resources and they are not dependable.

The Java service team does not recommend that you use finalizers for process control or for tidying Java resources. In fact, use finalizers as little as possible.

For tidying Java resources, consider the use of a cleanup routine. When you have finished with an object, call the routine to null out all references, deregister listeners, clear out hash tables, and other cleanup operation. Such a routine is far more efficient than using a finalizer and has the useful side-benefit of speeding up garbage collection. The Garbage Collector does not have so many object references to chase in the next garbage collection cycle.

Manually starting the Garbage Collector

Manually starting the Garbage Collector (GC) can degrade JVM performance.

See list item 4b in "Interaction of the Garbage Collector with applications" on page 46. The GC can honor a manual call; for example, through the System.gc() call. This call nearly always starts a garbage collection cycle, which is a heavy use of computer resources.

The Java service team recommends that this call is not used, or, if it is, it is enclosed in conditional statements that block its use in an application runtime environment. The GC is carefully adjusted to deliver maximum performance to the JVM. If you force it to run, you severely degrade JVM performance

The previous topics indicate that it is not sensible to try to force the GC to do something predictable, such as collecting your new garbage or running a finalizer. You cannot predict when the GC will act. Let the GC run inside the parameters that an application selects at startup time. This method nearly always produces best performance.

Several customer applications have been turned from unacceptable to acceptable performance by blocking out manual invocations of the GC. One enterprise application had more than four hundred System.gc() calls.

Frequently asked questions about the Garbage Collector

Examples of subjects that have answers in this section include default values, Garbage Collector (GC) policies, GC helper threads, Mark Stack Overflow, heap operation, and out of memory conditions.

What are the default heap and native stack sizes?

See "Default settings for the JVM" on page 474.

What is the difference between the GC policies `gencon`, `balanced`, `optavgpause`, and `optthruput`?

`gencon`

The **`gencon`** policy (default) uses a concurrent mark phase combined with generational garbage collection to help minimize the time that is spent in any garbage collection pause. This policy is particularly useful for applications with many short-lived objects, such as transactional applications. Pause times can be significantly shorter than with the **`optthruput`** policy, while still producing good throughput. Heap fragmentation is also reduced.

`balanced`

The **`balanced`** policy uses mark, sweep, compact and generational style garbage collection. The concurrent mark phase is disabled; concurrent garbage collection technology is used, but not in the way that concurrent mark is implemented for other policies. The **`balanced`** 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 tries to avoid global collections by matching object allocation and survival rates. If you have problems with application pause times that are caused by global garbage collections, particularly compactions, this policy might improve application performance. For more information about this policy, including when to use it, see "Balanced Garbage Collection policy" on page 39.

`optavgpause`

The **`optavgpause`** policy uses concurrent mark and concurrent sweep phases. Pause times are shorter than with **`optthruput`**, but application

throughput is reduced because some garbage collection work is taking place while the application is running. Consider using this policy if you have a large heap size (available on 64-bit platforms), because this policy limits the effect of increasing heap size on the length of the garbage collection pause. However, if your application uses many short-lived objects, the **gencon** policy might produce better performance.

subpool
The **subpool** policy is deprecated and is now an alias for **optthruput**. Therefore, if you use this option, the effect is the same as **optthruput**.

optthruput
The **optthruput** policy disables the concurrent mark phase. The application stops during global garbage collection, so long pauses can occur. This configuration is typically used for large-heap applications when high application throughput, rather than short garbage collection pauses, is the main performance goal. If your application cannot tolerate long garbage collection pauses, consider using another policy, such as **gencon**.

What is the default GC mode (gencon, optavgpause, or optthruput)?
gencon - that is, combined use of the generational collector and concurrent marking.

How many GC helper threads are created or "spawned"? What is their work?
The garbage collector creates *n*-1 helper threads, where *n* is the number of GC threads specified by the **-Xgcthreads**<*number*> option. See "Garbage Collector command-line options" on page 453 for more information. If you specify **-Xgcthreads1**, the garbage collector does not create any helper threads. Setting the **-Xgcthreads** option to a value that is greater than the number of processors on the system does not improve performance, but might alleviate mark-stack overflows, if your application suffers from them.

These helper threads work with the main GC thread during the following phases:

- Parallel mark phase
- Parallel bitwise sweep phase
- Parallel compaction phase
- Parallel scavenger phase
- Parallel copy-forward phase

What is Mark Stack Overflow (MSO)? Why is MSO bad for performance?
Work packets are used for tracing all object reference chains from the roots. Each such reference that is found is pushed onto the mark stack so that it can be traced later. The number of work packets allocated is based on the heap size and therefore is finite and can overflow. This situation is called Mark Stack Overflow (MSO). The algorithms to handle this situation are expensive in processing terms, and therefore MSO has a large impact on GC performance.

How can I prevent Mark Stack Overflow?
The following suggestions are not guaranteed to avoid MSO:

- Increase the number of GC helper threads using **-Xgcthreads** command-line option
- Decrease the size of the Java heap using the **-Xmx** setting.

- Use a small initial value for the heap or use the default.
- Reduce the number of objects the application allocates.
- If MSO occurs, you see entries in the verbose gc as follows:

  ```
  <warning details="work stack overflow" count="<mso_count>"
           packetcount="<allocated_packets>" />
  ```

  Where <mso_count> is the number of times MSO has occurred and <allocated_packets> is the number of work packets that were allocated. By specifying a larger number, say 50% more, with **-Xgcworkpackets<number>**, the likelihood of MSO can be reduced.

When and why does the Java heap expand?

The JVM starts with a small default Java heap, and it expands the heap based on the allocation requests made by an application until it reaches the value specified by **-Xmx**. Expansion occurs after GC if GC is unable to free enough heap storage for an allocation request. Expansion also occurs if the JVM determines that expanding the heap is required for better performance.

When does the Java heap shrink?

Heap shrinkage occurs when GC determines that there is heap storage space available, and releasing some heap memory is beneficial for system performance. Heap shrinkage occurs after GC, but when all the threads are still suspended.

Does GC guarantee that it clears all the unreachable objects?

GC guarantees only that all the objects that were not reachable at the beginning of the mark phase are collected. While running concurrently, our GC guarantees only that all the objects that were unreachable when concurrent mark began are collected. Some objects might become unreachable during concurrent mark, but they are not guaranteed to be collected.

I am getting an `OutOfMemoryError`. Does this mean that the Java heap is exhausted?

Not necessarily. Sometimes the Java heap has free space but an `OutOfMemoryError` can occur. The error might occur for several reasons:

- Shortage of memory for other operations of the JVM.
- Some other memory allocation failing. The JVM throws an `OutOfMemoryError` in such situations.
- Excessive memory allocation in other parts of the application, unrelated to the JVM, if the JVM is just a part of the process, rather than the entire process (JVM through JNI, for instance).
- The heap has been fully expanded, and an excessive amount of time (95%) is being spent in the GC. This check can be disabled using the option **-Xdisableexcessivegc**.

When I see an `OutOfMemoryError`, does that mean that the Java program exits?

Not always. Java programs can catch the exception thrown when `OutOfMemory` occurs, and (possibly after freeing up some of the allocated objects) continue to run.

In `verbose:gc` output, sometimes I see more than one GC for one allocation failure. Why?

You see this message when GC decides to clear all soft references. The GC

is called once to do the regular garbage collection, and might run again to clear soft references. Therefore, you might see more than one GC cycle for one allocation failure.

Class loading

The Java 2 JVM introduced a new class loading mechanism with a parent-delegation model. The parent-delegation architecture to class loading was implemented to aid security and to help programmers to write custom class loaders.

Class loading loads, verifies, prepares and resolves, and initializes a class from a Java class file.

- **Loading** involves obtaining the byte array representing the Java class file.
- **Verification** of a Java class file is the process of checking that the class file is structurally well-formed and then inspecting the class file contents to ensure that the code does not attempt to perform operations that are not permitted.
- **Preparation** involves the allocation and default initialization of storage space for static class fields. Preparation also creates method tables, which speed up virtual method calls, and object templates, which speed up object creation.
- **Initialization** involves the processing of the class's class initialization method, if defined, at which time static class fields are initialized to their user-defined initial values (if specified).

Symbolic references in a Java class file, such as to classes or object fields that reference a field's value, are resolved at run time to direct references only. This resolution might occur either:

- After preparation but before initialization
- Or, more typically, at some point following initialization, but before the first reference to that symbol.

The delay is generally to increase processing speed. Not all symbols in a class file are referenced during processing; by delaying resolution, fewer symbols might have to be resolved. The cost of resolution is gradually reduced over the total processing time.

The parent-delegation model

The delegation model requires that any request for a class loader to load a given class is first delegated to its parent class loader before the requested class loader tries to load the class itself. The parent class loader, in turn, goes through the same process of asking its parent. This chain of delegation continues through to the bootstrap class loader (also known as the primordial or system class loader). If a class loader's parent can load a given class, it returns that class. Otherwise, the class loader attempts to load the class itself.

The JVM has three class loaders, each possessing a different scope from which it can load classes. As you descend the hierarchy, the scope of available class repositories widens, and typically the repositories are less trusted:

```
Bootstrap
|
Extensions
|
Application
```

At the top of the hierarchy is the bootstrap class loader. This class loader is responsible for loading only the classes that are from the core Java API. These classes are the most trusted and are used to bootstrap the JVM.

The extensions class loader can load classes that are standard extensions packages in the extensions directory.

The application class loader can load classes from the local file system, and will load files from the CLASSPATH. The application class loader is the parent of any custom class loader or hierarchy of custom class loaders.

Because class loading is always delegated first to the parent of the class loading hierarchy, the most trusted repository (the core API) is checked first, followed by the standard extensions, then the local files that are on the class path. Finally, classes that are located in any repository that your own class loader can access, are accessible. This system prevents code from less-trusted sources from replacing trusted core API classes by assuming the same name as part of the core API.

Namespaces and the runtime package

Loaded classes are identified by both the class name and the class loader that loaded it. This separates loaded classes into namespaces that the class loader identifies.

A namespace is a set of class names that are loaded by a specific class loader. When an entry for a class has been added into a namespace, it is impossible to load another class of the same name into that namespace. Multiple copies of any given class can be loaded because a namespace is created for each class loader.

Namespaces cause classes to be segregated by class loader, thereby preventing less-trusted code loaded from the application or custom class loaders from interacting directly with more trusted classes. For example, the core API is loaded by the bootstrap class loader, unless a mechanism is specifically provided to allow them to interact. This prevents possibly malicious code from having guaranteed access to all the other classes.

You can grant special access privileges between classes that are in the same package by the use of package or protected access. This gives access rights between classes of the same package, but only if they were loaded by the same class loader. This stops code from an untrusted source trying to insert a class into a trusted package. As discussed earlier, the delegation model prevents the possibility of replacing a trusted class with a class of the same name from an untrusted source. The use of namespaces prevents the possibility of using the special access privileges that are given to classes of the same package to insert code into a trusted package.

Custom class loaders

You might want to write your own class loader so that you can load classes from an alternate repository, partition user code, or unload classes.

There are three main reasons why you might want to write your own class loader.

1. To allow class loading from alternative repositories.

   This is the most common case, in which an application developer might want to load classes from other locations, for example, over a network connection.
2. To partition user code.

This case is less frequently used by application developers, but widely used in servlet engines.

3. To allow the unloading of classes.

   This case is useful if the application creates large numbers of classes that are used for only a finite period. Because a class loader maintains a cache of the classes that it has loaded, these classes cannot be unloaded until the class loader itself has been dereferenced. For this reason, system and extension classes are never unloaded, but application classes can be unloaded when their class loader is.

For much more detailed information about the class loader, see http://www.ibm.com/developerworks/java/library/j-dclp1/. This article is the first in a series that helps you to write your own class loader.

Class data sharing

The class sharing feature in the IBM Version 7 SDK offers the transparent and dynamic sharing of data between multiple Java Virtual Machines (JVMs). When enabled, JVMs use shared memory to obtain and store data, including information about: loaded classes, Ahead-Of-Time (AOT) compiled code, commonly used UTF-8 strings, and Java Archive (JAR) file indexes.

This form of class sharing is an advancement on earlier JVMs that have offered some form of class sharing between multiple JVMs; for example, the IBM Persistent Reusable JVM on z/OS, Oracle Corporation "CDS" feature in their Java 5.0 release, and the bytecode verification cache in the i5/OS™ Classic VM.

You can enable shared classes with the **-Xshareclasses** command-line option. For reference information about **-Xshareclasses**, see “JVM command-line options” on page 428.

For diagnosing problems with shared classes, see “Shared classes diagnostic data” on page 344.

When loading a class, the JVM internally stores the class in two key parts:

- The immutable (read only) portion of the class.
- The mutable (writeable) portion of the class.

When enabled, shared classes shares the immutable parts of a class between JVMs, which has the following benefits:

- The amount of physical memory used can be significantly less when using more than one JVM instance.
- Loading classes from a populated cache is faster than loading classes from disk, because classes are already partially verified and are possibly already loaded in memory. Therefore, class sharing also benefits applications that regularly start new JVM instances doing similar tasks.

Caching AOT methods reduces the affect of JIT compilation when the same classes are loaded by multiple JVMs. In addition, because a shared classes cache might persist beyond the life of a JVM, subsequent JVMs that run can benefit from AOT methods already stored in the cache.

Key points to note about the IBM class sharing feature are:

- Class data sharing is available on all the platforms that IBM supports in Java V7, apart from the Oracle Solaris and HP hybrids.

- Classes are stored in a named “class cache”, which is either a memory-mapped file or an area of shared memory, allocated by the first JVM that needs to use it.
- A JVM can connect to a cache with either read-write or read-only access. Using read-only access allows greater isolation when many users are sharing the same cache.
- The class cache memory can be protected from accidental corruption using memory page protection.
- The JVM determines how many AOT methods get added to the cache and when. The amount of AOT method data is typically no more than 10% of the amount of class data cached.
- The JVM automatically removes and rebuilds a class cache if it decides that the cache is corrupted, or if the cache was created by a different level of service release.
- A separate, independent cache is created for each JVM version. For example, if you are running both 31-bit and 64-bit JVMs, two caches are created. If you are running Java 6 and Java 7, two caches are created.
- The file system location of the cache files can now be specified on the command line. Persistent cache files can be moved and copied around the file system. Persistent cache files can also be moved and copied between computers that use the same operating system and hardware.
- Filters can be applied to Java class loaders to allow users to limit the classes being shared.
- Any JVM can read from or update the cache, although a JVM can connect to only one cache at a time.
- The cache persists beyond the lifetime of any JVM connected to it until it is explicitly removed. Persistent caches (not available on z/OS) remain even after the operating system is shut down. Non-persistent caches are lost when the operating system is shut down.
- When a JVM loads a class, it looks first for the class in the cache to which it is connected and, if it finds the class it needs, it loads the class from the cache. Otherwise, it loads the class from disk and adds it to the cache where possible.
- When a cache becomes full, classes in the cache can still be shared, but no new data can be added.
- Because the class cache persists beyond the lifetime of any JVM connected to it, if changes are made to classes on the file system, some classes in the cache might become out-of-date (or “stale”). This situation is managed transparently; the updated version of the class is detected by the next JVM that loads it and the class cache is updated where possible.
- Sharing of retransformed and redefined bytecode is supported, but must be used with care.
- Access to the class data cache is protected by Java permissions if a security manager is installed.
- Classes generated using reflection cannot be shared.
- The classes cache stores class LineNumberTable and LocalVariableTable information in a reserved region of the cache during 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.

The JIT compiler

The Just-In-Time (JIT) compiler is a component of the Java Runtime Environment. It improves the performance of Java applications by compiling bytecodes to native machine code at run time. This section summarizes the relationship between the JVM and the JIT compiler and gives a short description of how the compiler works.

JIT compiler overview

The Just-In-Time (JIT) compiler is a component of the Java Runtime Environment that improves the performance of Java applications at run time.

Java programs consists of classes, which contain platform-neutral bytecodes that can be interpreted by a JVM on many different computer architectures. At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation. The additional processor and memory usage during interpretation means that a Java application performs more slowly than a native application. The JIT compiler helps improve the performance of Java programs by compiling bytecodes into native machine code at run time.

The JIT compiler is enabled by default, and is activated when a Java method is called. The JIT compiler compiles the bytecodes of that method into native machine code, compiling it "just in time" to run. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it. Theoretically, if compilation did not require processor time and memory usage, compiling every method could allow the speed of the Java program to approach that of a native application.

JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called. Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance.

In practice, methods are not compiled the first time they are called. For each method, the JVM maintains a call count, which is incremented every time the method is called. The JVM interprets a method until its call count exceeds a JIT compilation threshold. Therefore, often-used methods are compiled soon after the JVM has started, and less-used methods are compiled much later, or not at all. The JIT compilation threshold helps the JVM start quickly and still have improved performance. The threshold has been carefully selected to obtain an optimal balance between startup times and long term performance.

After a method is compiled, its call count is reset to zero and subsequent calls to the method continue to increment its count. When the call count of a method reaches a JIT recompilation threshold, the JIT compiler compiles it a second time, applying a larger selection of optimizations than on the previous compilation. This process is repeated until the maximum optimization level is reached. The busiest methods of a Java program are always optimized most aggressively, maximizing the performance benefits of using the JIT compiler. The JIT compiler can also measure operational data at run time, and use that data to improve the quality of further recompilations.

The JIT compiler can be disabled, in which case the entire Java program will be interpreted. Disabling the JIT compiler is not recommended except to diagnose or work around JIT compilation problems.

How the JIT compiler optimizes code

When a method is chosen for compilation, the JVM feeds its bytecodes to the Just-In-Time compiler (JIT). The JIT needs to understand the semantics and syntax of the bytecodes before it can compile the method correctly.

To help the JIT compiler analyze the method, its bytecodes are first reformulated in an internal representation called *trees*, which resembles machine code more closely than bytecodes. Analysis and optimizations are then performed on the trees of the method. At the end, the trees are translated into native code. The remainder of this section provides a brief overview of the phases of JIT compilation. For more information, see "JIT and AOT problem determination" on page 322.

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 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** option. For information on using this option, see "JIT and AOT command-line options" on page 448.

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

The compilation consists of the following phases:

1. Inlining
2. Local optimizations
3. Control flow optimizations
4. Global optimizations
5. Native code generation

All phases except native code generation are cross-platform code.

Phase 1 - inlining

Inlining is the process by which the trees of smaller methods are merged, or "inlined", into the trees of their callers. This speeds up frequently executed method calls.

Two inlining algorithms with different levels of aggressiveness are used, depending on the current optimization level. Optimizations performed in this phase include:

- Trivial inlining
- Call graph inlining
- Tail recursion elimination
- Virtual call guard optimizations

Phase 2 - local optimizations

Local optimizations analyze and improve a small section of the code at a time. Many local optimizations implement tried and tested techniques used in classic static compilers.

The optimizations include:

- Local data flow analyses and optimizations

- Register usage optimization
- Simplifications of Java idioms

These techniques are applied repeatedly, especially after global optimizations, which might have pointed out more opportunities for improvement.

Phase 3 - control flow optimizations

Control flow optimizations analyze the flow of control inside a method (or specific sections of it) and rearrange code paths to improve their efficiency.

The optimizations are:

- Code reordering, splitting, and removal
- Loop reduction and inversion
- Loop striding and loop-invariant code motion
- Loop unrolling and peeling
- Loop versioning and specialization
- Exception-directed optimization
- Switch analysis

Phase 4 - global optimizations

Global optimizations work on the entire method at once. They are more "expensive", requiring larger amounts of compilation time, but can provide a great increase in performance.

The optimizations are:

- Global data flow analyses and optimizations
- Partial redundancy elimination
- Escape analysis
- GC and memory allocation optimizations
- Synchronization optimizations

Phase 5 - native code generation

Native code generation processes vary, depending on the platform architecture. Generally, during this phase of the compilation, the trees of a method are translated into machine code instructions; some small optimizations are performed according to architecture characteristics.

The compiled code is placed into a part of the JVM process space called the *code cache*; the location of the method in the code cache is recorded, so that future calls to it will call the compiled code. At any given time, the JVM process consists of the JVM executable files and a set of JIT-compiled code that is linked dynamically to the bytecode interpreter in the JVM.

Frequently asked questions about the JIT compiler

Examples of subjects that have answers in this section include disabling the JIT compiler, use of alternative JIT compilers, control of JIT compilation and dynamic control of the JIT compiler.

Can I disable the JIT compiler?

Yes. The JIT compiler is turned on by default, but you can turn it off with the appropriate command-line parameter. For more information, see “Disabling the JIT or AOT compiler” on page 323.

Can I use another vendor's JIT compiler?
No.

Can I use any version of the JIT compiler with the JVM?
No. The two are tightly coupled. You must use the version of the JIT compiler that comes with the JVM package that you use.

Can the JIT compiler "decompile" methods?
No. After a method is compiled by the JIT compiler, the native code is used instead for the remainder of the execution of the program. An exception to this rule is a method in a class that was loaded with a custom (user-written) class loader, which has since been unloaded (garbage-collected). In fact, when a class loader is garbage-collected, the compiled methods of all classes that are loaded by that class loader are discarded.

Can I control the JIT compilation?
Yes. For more information, see "JIT and AOT problem determination" on page 322. In addition, advanced diagnostic settings are available to IBM engineers.

Can I dynamically control the JIT compiler?
No. You can pass options to the JIT compiler to modify the behavior, but only at JVM startup time, because the JIT compiler is started up at the same time as the JVM. However, a Java program can use the java.lang.Compiler API to enable and disable the JIT compiler at run time.

How much memory does the code cache consume?
The JIT compiler uses memory intelligently. When the code cache is initialized, it consumes relatively little memory. As more methods are compiled into native code, the code cache grows dynamically to accommodate the needs of the program. Space that is previously occupied by discarded or recompiled methods is reclaimed and reused. When the size of the code cache reaches a predefined maximum limit, it stops growing. At this point, the JIT compiler stops compiling methods to avoid exhausting the system memory and affecting the stability of the application or the operating system.

The AOT compiler

Ahead-Of-Time (AOT) compilation allows the compilation of Java classes into native code for subsequent executions of the same program. The AOT compiler works with the class data sharing framework.

The AOT compiler generates native code dynamically while an application runs and caches any generated AOT code in the shared data cache. Subsequent JVMs that execute the method can load and use the AOT code from the shared data cache without incurring the performance decrease experienced with JIT-compiled native code.

The AOT compiler is enabled by default, but is only active when shared classes are enabled. By default, shared classes are disabled so that no AOT activity occurs. When the AOT compiler is active, the compiler selects the methods to be AOT compiled with the primary goal of improving startup time.

Note: Because AOT code must persist over different program executions, AOT-generated code does not perform as well as JIT-generated code. AOT code usually performs better than interpreted code.

In a JVM without an AOT compiler or with the AOT compiler disabled, the JIT compiler selectively compiles frequently used methods into optimized native code.

There is a time cost associated with compiling methods because the JIT compiler operates while the application is running. Because methods begin by being interpreted and most JIT compilations occur during startup, startup times can be increased.

Startup performance can be improved by using the shared AOT code to provide native code without compiling. There is a small time cost to load the AOT code for a method from the shared data cache and bind it into a running program. The time cost is low compared to the time it takes the JIT compiler to compile that method.

The **-Xshareclasses** option can be used to enable shared classes, which might also activate the AOT compiler if AOT is enabled.

Java Remote Method Invocation

Java Remote Method Invocation (Java RMI) enables you to create distributed Java technology-based applications that can communicate with other such applications. Methods of remote Java objects can be run from other Java virtual machines (JVMs), possibly on different hosts.

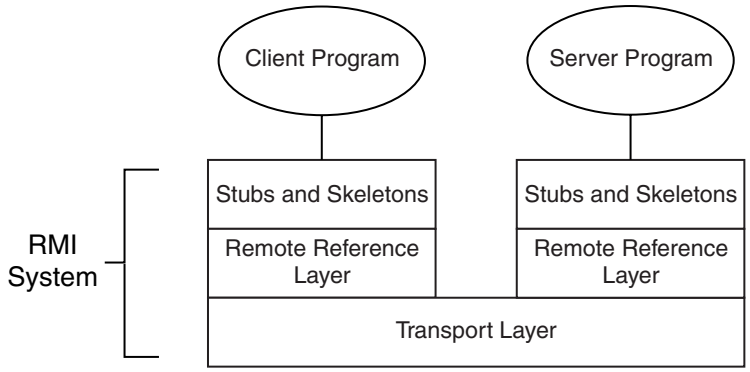
RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, supporting object-oriented polymorphism. The RMI registry is a lookup service for ports.

The RMI implementation

The RMI implementation consists of three abstraction layers.

These abstraction layers are:

1. The **Stub and Skeleton** layer, which intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.
2. The **Remote Reference** layer understands how to interpret and manage references made from clients to the remote service objects.
3. The bottom layer is the **Transport** layer, which is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.



On top of the TCP/IP layer, RMI uses a wire-level protocol called Java Remote Method Protocol (JRMP), which works like this:

1. Objects that require remote behavior should extend the RemoteObject class, typically through the UnicastRemoteObject subclass.
   a. The UnicastRemoteObject subclass exports the remote object to make it available for servicing incoming RMI calls.
   b. Exporting the remote object creates a new server socket, which is bound to a port number.
   c. A thread is also created that listens for connections on that socket. The server is registered with a registry.
   d. A client obtains details of connecting to the server from the registry.
   e. Using the information from the registry, which includes the hostname and the port details of the server's listening socket, the client connects to the server.
2. When the client issues a remote method invocation to the server, it creates a TCPConnection object, which opens a socket to the server on the port specified and sends the RMI header information and the marshalled arguments through this connection using the StreamRemoteCall class.
3. On the server side:
   a. When a client connects to the server socket, a new thread is assigned to deal with the incoming call. The original thread can continue listening to the original socket so that additional calls from other clients can be made.
   b. The server reads the header information and creates a RemoteCall object of its own to deal with unmarshalling the RMI arguments from the socket.
   c. The serviceCall() method of the Transport class services the incoming call by dispatching it
   d. The dispatch() method calls the appropriate method on the object and pushes the result back down the wire.
   e. If the server object throws an exception, the server catches it and marshals it down the wire instead of the return value.
4. Back on the client side:
   a. The return value of the RMI is unmarshalled and returned from the stub back to the client code itself.
   b. If an exception is thrown from the server, that is unmarshalled and thrown from the stub.

Thread pooling for RMI connection handlers

When a client connects to the server socket, a new thread is forked to deal with the incoming call. The IBM SDK implements thread pooling in the sun.rmi.transport.tcp.TCPTransport class.

Thread pooling is not enabled by default. Enable it with this command-line setting:

```
-Dsun.rmi.transport.tcp.connectionPool=true
```

Alternatively, you could use a non-null value instead of `true`.

With the connectionPool enabled, threads are created only if there is no thread in the pool that can be reused. In the current implementation of the connection Pool, the RMI connectionHandler threads are added to a pool and are never removed. Enabling thread pooling is not recommended for applications that have only limited RMI usage. Such applications have to live with these threads during the RMI off-peak times as well. Applications that are mostly RMI intensive can benefit

by enabling the thread pooling because the connection handlers will be reused, avoiding the additional memory usage when creating these threads for every RMI call.

Understanding distributed garbage collection

The RMI subsystem implements reference counting based Distributed Garbage Collection (DGC) to provide automatic memory management facilities for remote server objects.

When the client creates (unmarshalls) a remote reference, it calls dirty() on the server-side DGC. After the client has finished with the remote reference, it calls the corresponding clean() method.

A reference to a remote object is leased for a time by the client holding the reference. The lease period starts when the dirty() call is received. The client must renew the leases by making additional dirty() calls on the remote references it holds before such leases expire. If the client does not renew the lease before it expires, the distributed garbage collector assumes that the remote object is no longer referenced by that client.

DGCClient implements the client side of the RMI distributed garbage collection system. The external interface to DGCClient is the registerRefs() method. When a LiveRef to a remote object enters the JVM, it must be registered with the DGCClient to participate in distributed garbage collection. When the first LiveRef to a particular remote object is registered, a dirty() call is made to the server-side DGC for the remote object. The call returns a lease guaranteeing that the server-side DGC will not collect the remote object for a certain time. While LiveRef instances to remote objects on a particular server exist, the DGCClient periodically sends more dirty calls to renew its lease. The DGCClient tracks the local availability of registered LiveRef instances using phantom references. When the LiveRef instance for a particular remote object is garbage collected locally, a clean() call is made to the server-side DGC. The call indicates that the server does not need to keep the remote object alive for this client. The RenewCleanThread handles the asynchronous client-side DGC activity by renewing the leases and making clean calls. So this thread waits until the next lease renewal or until any phantom reference is queued for generating clean requests as necessary.

Debugging applications involving RMI

When debugging applications involving RMI you need information on exceptions and properties settings, solutions to common problems, answers to frequently asked questions, and useful tools.

The list of exceptions that can occur when using RMI and their context is included in the *RMI Specification* document at: http://download.oracle.com/javase/7/docs/platform/rmi/spec/rmi-exceptions.html#3601

Properties settings that are useful for tuning, logging, or tracing RMI servers and clients can be found at: http://download.oracle.com/javase/7/docs/technotes/guides/rmi/javarmiproperties.html

Solutions to some common problems and answers to frequently asked questions related to RMI and object serialization can be found at: http://download.oracle.com/javase/7/docs/technotes/guides/rmi/faq.html

Network monitoring tools like netstat and tcpdump are useful for debugging RMI problems because they enable you to see the traffic at the network level.

The ORB

This description of the Object Request Broker (ORB) provides background information to help you diagnose problems with the ORB.

The topics in this chapter are:

- "CORBA"
- "RMI and RMI-IIOP" on page 65
- "Java IDL or RMI-IIOP?" on page 65
- "RMI-IIOP limitations" on page 66
- "Further reading" on page 66
- "Examples of client–server applications" on page 66

The IBM ORB ships with the JVM and is used by the IBM WebSphere Application Server. It is one of the enterprise features of the Java 2 Standard Edition. The ORB is both a tool and a runtime component. It provides distributed computing through the CORBA Internet Inter-Orb Protocol (IIOP) communication protocol. The protocol is defined by the Object Management Group (OMG). The ORB runtime environment consists of a Java implementation of a CORBA ORB. The ORB toolkit provides APIs and tools for both the Remote Method Invocation (RMI) programming model and the Interface Definition Language (IDL) programming model.

Note: The use of the Channel Framework ORB transport mode is deprecated in this release of the IBM SDK.

CORBA

The Common Object Request Broker Architecture (CORBA) is an open, vendor-independent specification for distributed computing. It is published by the Object Management Group (OMG).

Most applications need different objects on various platforms and operating systems to communicate with each other across networks. CORBA enables objects to interoperate in this way, using the Internet Inter-ORB Protocol (IIOP). To help objects understand the operations available, and the syntax required to invoke them, an Interface Definition Language (IDL) is used. The IDL is programming-language independent, to increase the interoperability between objects.

When an application developer defines an object, they also define other aspects. The aspects include the position of the object in an overall hierarchy, object attributes, and possible operations. Next, the aspects are all described in the IDL. The description is then converted into an implementation by using an IDL compiler. For example, IDLJ is an IDL compiler for the Java language, and converts an IDL description into a Java source code. The benefit of this is that the object implementation is "encapsulated" by the IDL definition. Any other objects wanting to interoperate can do so using mechanisms defined using the shared IDL.

Developers enable this interoperability by defining the hierarchy, attributes, and operations of objects using IDL. They then use an IDL compiler (such as IDLJ for Java) to map the definition onto an implementation in a programming language. The implementation of an object is encapsulated. Clients of the object can see only its external IDL interface. The OMG has produced specifications for mappings from IDL to many common programming languages, including C, C++, and Java

An essential part of the CORBA specification is the Object Request Broker (ORB). The ORB routes requests from a client object to a remote object. The ORB then returns any responses to the required destinations. Java contains an implementation of the ORB that communicates by using IIOP.

RMI and RMI-IIOP

This description compares the two types of remote communication in Java; Remote Method Invocation (RMI) and RMI-IIOP.

RMI is Java's traditional form of remote communication. It is an object-oriented version of Remote Procedure Call (RPC). It uses the nonstandardized Java Remote Method Protocol (JRMP) to communicate between Java objects. This provides an easy way to distribute objects, but does not allow for interoperability between programming languages.

RMI-IIOP is an extension of traditional Java RMI that uses the IIOP protocol. This protocol allows RMI objects to communicate with CORBA objects. Java programs can therefore interoperate transparently with objects that are written in other programming languages, provided that those objects are CORBA-compliant. Objects can still be exported to traditional RMI (JRMP) and the two protocols can communicate.

A terminology difference exists between the two protocols. In RMI (JRMP), the server objects are called skeletons; in RMI-IIOP, they are called ties. Client objects are called stubs in both protocols.

Java IDL or RMI-IIOP?

There are circumstances in which you might choose to use RMI-IIOP and others in which you might choose to use Java IDL.

RMI-IIOP is the method that is chosen by Java programmers who want to use the RMI interfaces, but use IIOP as the transport. RMI-IIOP requires that all remote interfaces are defined as Java RMI interfaces. Java IDL is an alternative solution, intended for CORBA programmers who want to program in Java to implement objects that are defined in IDL. The general rule that is suggested by Oracle is to use Java IDL when you are using Java to access existing CORBA resources, and RMI-IIOP to export RMI resources to CORBA.

RMI-IIOP limitations

You must understand the limitations of RMI-IIOP when you develop an RMI-IIOP application, and when you deploy an existing CORBA application in a Java-IIOP environment.

In a Java-only application, RMI (JRMP) is more lightweight and efficient than RMI-IIOP, but less scalable. Because it has to conform to the CORBA specification for interoperability, RMI-IIOP is a more complex protocol. Developing an RMI-IIOP application is much more similar to CORBA than it is to RMI (JRMP).

You must take care if you try to deploy an existing CORBA application in a Java RMI-IIOP environment. An RMI-IIOP client cannot necessarily access every existing CORBA object. The semantics of CORBA objects that are defined in IDL are a superset of those of RMI-IIOP objects. That is why the IDL of an existing CORBA object cannot always be mapped into an RMI-IIOP Java interface. It is only when the semantics of a specific CORBA object are designed to relate to those of RMI-IIOP that an RMI-IIOP client can call a CORBA object.

Further reading

There are links to CORBA specifications, CORBA basics, and the RMI-IIOP Programmer's Guide.

Object Management Group Web site: http://www.omg.org contains CORBA specifications that are available to download.

OMG - CORBA Basics: http://www.omg.org/gettingstarted/corbafaq.htm. Remember that some features discussed here are not implemented by all ORBs.

Examples of client–server applications

CORBA, RMI (JRMP), and RMI-IIOP approaches are used to present three client-server example applications. All the applications use the RMI-IIOP IBM ORB.

Interfaces

The interfaces to be implemented are CORBA IDL and Java RMI.

The two interfaces are:

- CORBA IDL Interface (`Sample.idl`):

  ```
  interface Sample { string message(); };
  ```

- Java RMI Interface (`Sample.java`):

  ```
  public interface Sample extends java.rmi.Remote
  { public String message() throws java.rmi.RemoteException; }
  ```

These two interfaces define the characteristics of the remote object. The remote object implements a method, named message. The method does not need any parameter, and it returns a string. For further information about IDL and its mapping to Java, see the OMG specifications (http://www.omg.org).

Remote object implementation (or servant)

This description shows possible implementations of the object.

The possible RMI(JRMP) and RMI-IIOP implementations (`SampleImpl.java`) of this object could be:

```
 public class SampleImpl extends javax.rmi.PortableRemoteObject implements Sample {
       public SampleImpl() throws java.rmi.RemoteException { super(); }
       public String message() { return "Hello World!"; }
}
```

You can use the class PortableRemoteObject for both RMI over JRMP and IIOP. The effect is to make development of the remote object effectively independent of the protocol that is used. The object implementation does not need to extend PortableRemoteObject, especially if it already extends another class (single-class inheritance). Instead, the remote object instance must be exported in the server implementation. Exporting a remote object makes the object available to accept incoming remote method requests. When you extend javax.rmi.PortableRemoteObject, your class is exported automatically on creation.

The CORBA or Java IDL implementation of the remote object (servant) is:

```
public class SampleImpl extends _SamplePOA {
    public String message() { return "Hello World"; }
}
```

The POA is the Portable Object Adapter, described in "Portable object adapter" on page 82.

The implementation conforms to the Inheritance model, in which the servant extends directly the IDL-generated skeleton SamplePOA. You might want to use the Tie or Delegate model instead of the typical Inheritance model if your implementation must inherit from some other implementation. In the Tie model, the servant implements the IDL-generated operations interface (such as SampleOperations). The Tie model introduces a level of indirection, so that one extra method call occurs when you invoke a method. The server code describes the extra work that is required in the Tie model, so that you can decide whether to use the Tie or the Delegate model. In RMI-IIOP, you can use only the Tie or Delegate model.

Stubs and ties generation

The RMI-IIOP code provides the tools to generate stubs and ties for whatever implementation exists of the client and server.

The following table shows what command to run to get the stubs and ties (or skeletons) for each of the three techniques:

CORBA	RMI(JRMP)	RMI-IIOP
`idlj Sample.idl`	`rmic SampleImpl`	`rmic -iiop Sample`

Compilation generates the files that are shown in the following table. To keep the intermediate `.java` files, run the **`rmic`** command with the **`-keep`** option.

CORBA	RMI(JRMP)	RMI-IIOP
`Sample.java`	`SampleImpl_Skel.class`	`_SampleImpl_Tie.class`
`SampleHolder.java`	`SampleImpl_Stub.class`	`_Sample_Stub.class`
`SampleHelper.java`	`Sample.class` (`Sample.java` present)	`Sample.class` (`Sample.java` present)
`SampleOperations.java`	`SampleImpl.class` (only compiled)	`SampleImpl.class` (only compiled)
`_SampleStub.java`		

CORBA	RMI(JRMP)	RMI-IIOP
SamplePOA.java (**-fserver**, **-fall**, **-fserverTie**, **-fallTie**)		
SamplePOATie.java (**-fserverTie**, **-fallTie**)		
_SampleImplBase.java (**-oldImplBase**)		

Since the Java v1.4 ORB, the default object adapter (see the OMG CORBA specification v.2.3) is the Portable Object Adapter (POA). Therefore, the default skeletons and ties that the IDL compiler generates can be used by a server that is using the POA model and interfaces. By using the **idlj -oldImplBase** option, you can generate older versions of the server-side skeletons that are compatible with servers that are written in Java v1.3 and earlier.

Server code

The server application has to create an instance of the remote object and publish it in a naming service. The Java Naming and Directory Interface (JNDI) defines a set of standard interfaces. The interfaces are used to query a naming service, or to bind an object to that service.

The implementation of the naming service can be a CosNaming service in the Common Object Request Broker Architecture (CORBA) environment. A CosNaming service is a collection of naming services, and implemented as a set of interfaces defined by CORBA. Alternatively, the naming service can be implemented using a Remote Method Invocation (RMI) registry for an RMI(JRMP) application. You can use JNDI in CORBA and in RMI cases. The effect is to make the server implementation independent of the naming service that is used. For example, you could use the following code to obtain a naming service and bind an object reference in it:

```
Context ctx = new InitialContext(...);  // get hold of the initial context
ctx.bind("sample", sampleReference);  // bind the reference to the name "sample"
Object obj = ctx.lookup("sample");  // obtain the reference
```

To tell the application which naming implementation is in use, you must set one of the following Java properties:

java.naming.factory.initial
: Defined also as javax.naming.Context.INITIAL_CONTEXT_FACTORY, this property specifies the class name of the initial context factory for the naming service provider. For RMI registry, the class name is com.sun.jndi.rmi.registry.RegistryContextFactory. For the CosNaming Service, the class name is com.sun.jndi.cosnaming.CNCtxFactory.

java.naming.provider.url
: This property configures the root naming context, the Object Request Broker (ORB), or both. It is used when the naming service is stored in a different host, and it can take several URI schemes:
  - rmi
  - corbaname
  - corbaloc
  - IOR
  - iiop
  - iiopname

For example:

```
rmi://[<host>[:<port>]][/<initial_context>] for RMI registry
iiop://[<host>[:<port>]][/<cosnaming_name>] for COSNaming
```

To get the previous properties in the environment, you could code:

```
Hashtable env = new Hashtable();
Env.put(Context.INITIAL_CONTEXT_FACTORY,
       "com.sun.jndi.cosnaming.CNCtxFactory");
```

and pass the hash table as an argument to the constructor of InitialContext.

For example, with RMI(JRMP), you create an instance of the servant then follow the previous steps to bind this reference in the naming service.

With CORBA (Java IDL), however, you must do some extra work because you have to create an ORB. The ORB has to make the servant reference available for remote calls. This mechanism is typically controlled by the object adapter of the ORB.

```
public class Server {
 public static void main (String args []) {
  try {
   ORB orb = ORB.init(args, null);

   // Get reference to the root poa & activate the POAManager
   POA poa = (POA)orb.resolve_initial_references("RootPOA");
   poa.the_POAManager().activate();

   // Create a servant and register with the ORB
   SampleImpl sample = new SampleImpl();
   sample.setORB(orb);

   // TIE model ONLY
   // create a tie, with servant being the delegate and
   // obtain the reference ref for the tie
   SamplePOATie tie = new SamplePOATie(sample, poa);
   Sample ref = tie._this(orb);

   // Inheritance model ONLY
   // get object reference from the servant
   org.omg.CORBA.Object ref = poa.servant_to_reference(sample);
      Sample ref = SampleHelper.narrow(ref);

   // bind the object reference ref to the naming service using JNDI
      ..........(see previous code) .....
   orb.run();
  }
  catch(Exception e) {}
 }
}
```

For RMI-IIOP:

```
public class Server {
 public static void main (String args []) {
  try {
   ORB orb = ORB.init(args, null);

   // Get reference to the root poa & activate the POAManager
   POA poa = (POA)orb.resolve_initial_references("RootPOA");
   poa.the_POAManager().activate();

   // Create servant and its tie
   SampleImpl sample = new SampleImpl();
   _SampleImpl_Tie tie = (_SampleImpl_Tie)Util.getTie(sample);
```

```
   // get an usable object reference
   org.omg.CORBA.Object ref = poa.servant_to_reference((Servant)tie);

   // bind the object reference ref to the naming service using JNDI
      ..........(see previous code) .....
  }
  catch(Exception e) {}
 }
}
```

To use the previous Portable Object Adapter (POA) server code, you must use the **-iiop -poa** options together to enable **rmic** to generate the tie. If you do not use the POA, the RMI(IIOP) server code can be reduced to instantiating the servant (`SampleImpl sample = new SampleImpl()`). You then bind the servant to a naming service as is typically done in the RMI(JRMP) environment. In this case, you need use only the **-iiop** option to enable **rmic** to generate the RMI-IIOP tie. If you omit **-iiop**, the RMI(JRMP) skeleton is generated.

When you export an RMI-IIOP object on your server, you do not necessarily have to choose between JRMP and IIOP. If you need a single server object to support JRMP and IIOP clients, you can export your RMI-IIOP object to JRMP and to IIOP simultaneously. In RMI-IIOP terminology, this action is called *dual export*.

RMI Client example:

```
public class SampleClient {
  public static void main(String [] args) {
    try{
      Sample sampleref
      //Look-up the naming service using JNDI and get the reference
            .........
      // Invoke method
      System.out.println(sampleRef.message());
    }
    catch(Exception e) {}
  }
}
```

CORBA Client example:

```
public class SampleClient {
 public static void main (String [] args) {
  try {
   ORB orb = ORB.init(args, null);
   // Look up the naming service using JNDI
   ......
   // Narrowing the reference to the correct class
   Sample sampleRef = SampleHelper.narrow(o);
   // Method Invocation
   System.out.println(sampleRef.message());
  }
  catch(Exception e) {}
 }
}
```

RMI-IIOP Client example:

```
public class SampleClient {
 public static void main (String [] args) {
  try{
   ORB orb = ORB.init(args, null);
   // Retrieving reference from naming service
   ........
   // Narrowing the reference to the correct class
```

```
   Sample sampleRef = (Sample)PortableRemoteObject.narrow(o, Sample.class);
   // Method Invocation
   System.out.println(sampleRef.message());
  }
  catch(Exception e) {}
 }
}
```

Summary of major differences between RMI (JRMP) and RMI-IIOP

There are major differences in development procedures between RMI (JRMP) and RMI-IIOP. The points discussed here also represent work items that are necessary when you convert RMI (JRMP) code to RMI-IIOP code.

Because the usual base class of RMI-IIOP servers is PortableRemoteObject, you must change this import statement accordingly, in addition to the derivation of the implementation class of the remote object. After completing the Java coding, you must generate a tie for IIOP by using the rmic compiler with the **-iiop** option. Next, run the CORBA CosNaming tnameserv as a name server instead of rmiregistry.

For CORBA clients, you must also generate IDL from the RMI Java interface by using the rmic compiler with the **-idl** option.

All the changes in the import statements for server development apply to client development. In addition, you must also create a local object reference from the registered object name. The lookup() method returns a java.lang.Object, and you must then use the narrow() method of PortableRemoteObject to cast its type. You generate stubs for IIOP using the rmic compiler with the **-iiop** option.

Summary of differences in server development:

There are a number of differences in server development.

- Import statement:
  ```
  import javax.rmi.PortableRemoteObject;
  ```
- Implementation class of a remote object:
  ```
  public class SampleImpl extends PortableRemoteObject implements Sample
  ```
- Name registration of a remote object:
  ```
  NamingContext.rebind("Sample",ObjRef);
  ```
- Generate a tie for IIOP using the command:
  ```
  rmic -iiop
  ```
- Run **tnameserv** as a name server.
- Generate IDL for CORBA clients using the command:
  ```
  rmic -idl
  ```

Summary of differences in client development:

There are a number of differences in client development.

- Import statement:
  ```
  import javax.rmi.PortableRemoteObject;
  ```
- Identify a remote object by name:
  ```
  Object obj = ctx.lookup("Sample")

  MyObject myobj = (MyObject)PortableRemoteObject.narrow(obj,MyObject.class);
  ```
- Generate a stub for IIOP using the command:

```
rmic -iiop
```

Using the ORB

To use the Object Request Broker (ORB) effectively, you must understand the properties that the ORB contains. These properties change the behavior of the ORB.

The property values are listed as follows. All property values are specified as strings.

- **com.ibm.CORBA.AcceptTimeout**: (range: 0 through 5000) (default: 0=infinite timeout)

  The maximum number of milliseconds for which the ServerSocket waits in a call to `accept()`. If this property is not set, the default 0 is used. If it is not valid, 5000 is used.

- **com.ibm.CORBA.AllowUserInterrupt**:

  Set this property to `true` so that you can call `Thread.interrupt()` on a thread that is currently involved in a remote method call. The result is to stop that thread waiting for the call to return. Interrupting a call in this way causes a RemoteException to be thrown, containing a `CORBA.NO_RESPONSE` runtime exception with the `RESPONSE_INTERRUPTED` minor code.

  If this property is not set, the default behavior is to ignore any Thread.interrupt() received while waiting for a call to finish.

- **com.ibm.CORBA.ConnectTimeout**: (range: 0 through 300) (default: 0=infinite timeout)

  The maximum number of seconds that the ORB waits when opening a connection to another ORB. By default, no timeout is specified.

- **com.ibm.CORBA.BootstrapHost**:

  The value of this property is a string. This string can be the host name or the IP address of the host, such as 9.5.88.112. If this property is not set, the local host is retrieved by calling one of the following methods:

  – For applications: `InetAddress.getLocalHost().getHostAddress()`

  – For applets: `<applet>.getCodeBase().getHost()`

  The host name is the name of the system on which the initial server contact for this client is installed.

  Note: This property is deprecated. It is replaced by **-ORBInitRef** and **-ORBDefaultInitRef**.

- **com.ibm.CORBA.BootstrapPort**: (range: 0 through 2147483647=Java max int) (default: 2809)

  The port of the system on which the initial server contact for this client is listening.

  Note: This property is deprecated. It is replaced by **-ORBInitRef** and **-ORBDefaultInitRef**.

- **com.ibm.CORBA.BufferSize**: (range: 0 through 2147483647=Java max int) (default: 2048)

  The number of bytes of a General Inter-ORB Protocol (GIOP) message that is read from a socket on the first attempt. A larger buffer size increases the probability of reading the whole message in one attempt. Such an action might improve performance. The minimum size used is 24 bytes.

- **com.ibm.CORBA.ConnectionMultiplicity**: (range: 0 through 2147483647) (default: 1)

Setting this value to a number *n* greater than one causes a client ORB to multiplex communications to each server ORB. There can be no more than *n* concurrent sockets to each server ORB at any one time. This value might increase throughput under certain circumstances, particularly when a long-running, multithreaded process is acting as a client. The number of parallel connections can never exceed the number of requesting threads. The number of concurrent threads is therefore a sensible maximum limit for this property.

- **com.ibm.CORBA.enableLocateRequest**: (default: false)

  If this property is set, the ORB sends a LocateRequest before the actual Request.

- **com.ibm.CORBA.FragmentSize**: (range: 0 through 2147483647=Java max int) (default:1024)

  Controls GIOP 1.2 fragmentation. The size specified is rounded down to the nearest multiple of 8, with a minimum size of 64 bytes. You can disable message fragmentation by setting the value to 0.

- **com.ibm.CORBA.FragmentTimeout**: (range: 0 through 600000 ms) (default: 300000)

  The maximum length of time for which the ORB waits for second and subsequent message fragments before timing out. Set this property to 0 if timeout is not required.

- **com.ibm.CORBA.GIOPAddressingDisposition**: (range: 0 through 2) (default: 0)

  When a GIOP 1.2 Request, LocateRequest, Reply, or LocateReply is created, the addressing disposition is set depending on the value of this property:

  – 0 = Object Key
  – 1 = GIOP Profile
  – 2 = full IOR

  If this property is not set or is passed an invalid value, the default 0 is used.

- **com.ibm.CORBA.InitialReferencesURL**:

  The format of the value of this property is a correctly formed URL; for example, `http://w3.mycorp.com/InitRefs.file`. The actual file contains a name and value pair like: `NameService=<stringified_IOR>`. If you specify this property, the ORB does not attempt the bootstrap approach. Use this property if you do not have a bootstrap server and want to have a file on the webserver that serves the purpose.

  Note: This property is deprecated.

- **com.ibm.CORBA.ListenerPort**: (range: 0 through 2147483647=Java max int) (default: next available system assigned port number)

  The port on which this server listens for incoming requests. If this property is specified, the ORB starts to listen during `ORB.init()`.

- **com.ibm.CORBA.LocalHost**:

  The value of this property is a string. This string can be a host name or the IP address (ex. 9.5.88.112). If this property is not set, retrieve the local host by calling: `InetAddress.getLocalHost().getHostAddress()`. This property represents the host name (or IP address) of the system on which the ORB is running. The local host name is used by the server-side ORB to place the host name of the server into the IOR of a remote-able object.

- **com.ibm.CORBA.LocateRequestTimeout**: (range: 0 through 2147483647) (default: 0=infinity)

  Defines the number of seconds to wait before timing out on a LocateRequest message.

- **com.ibm.CORBA.MaxOpenConnections**: (range: 0 through 2147483647) (default: 240)

  Determines the maximum number of in-use connections that are to be kept in the connection cache table at any one time.
- **com.ibm.CORBA.MinOpenConnections**: (range: 0 through 2147483647) (default: 100)

  The ORB cleans up only connections that are not busy from the connection cache table, if the size is of the table is higher than the MinOpenConnections.
- **com.ibm.CORBA.NoLocalInterceptors**: (default: false)

  If this property is set to true, no local portable interceptors are used. The expected result is improved performance if interceptors are not required when connecting to a co-located object.
- **com.ibm.CORBA.ORBCharEncoding**: (default: ISO8859_1)

  Specifies the native encoding set used by the ORB for character data.
- **com.ibm.CORBA.ORBWCharDefault**: (default: UCS2)

  Indicates that wchar code set UCS2 is to be used with other ORBs that do not publish a wchar code set.
- **com.ibm.CORBA.RequestTimeout**: (range: 0 through 2147483647) (default: 0=infinity)

  Defines the number of seconds to wait before timing out on a Request message.
- **com.ibm.CORBA.SendingContextRunTimeSupported**: (default: true)

  Set this property to false to disable the CodeBase SendingContext RunTime service. This means that the ORB does not attach a SendingContextRunTime service context to outgoing messages.
- **com.ibm.CORBA.SendVersionIdentifier**: (default: false)

  Tells the ORB to send an initial dummy request before it starts to send any real requests to a remote server. This action determines the partner version of the remote server ORB, based on the response from that ORB.
- **com.ibm.CORBA.ServerSocketQueueDepth**: (range: 50 through 2147483647) (default: 0)

  The maximum queue length for incoming connection indications. A connect indication is a request to connect. If a connection indication arrives when the queue is full, the connection is refused. If the property is not set, the default 0 is used. If the property is not valid, 50 is used.
- **com.ibm.CORBA.ShortExceptionDetails**: (default: false)

  When a CORBA SystemException reply is created, the ORB, by default, includes the Java stack trace of the exception in an associated ExceptionDetailMessage service context. If you set this property to any value, the ORB includes a `toString` of the Exception instead.
- **com.ibm.tools.rmic.iiop.Debug**: (default: false)

  The **`rmic`** tool automatically creates import statements in the classes that it generates. If set to true, this property causes **`rmic`** to report the mappings of fully qualified class names to short names.
- **com.ibm.tools.rmic.iiop.SkipImports**: (default: false)

  If this property is set to true, classes are generated with **`rmic`** using fully qualified names only.
- **org.omg.CORBA.ORBId**

  Uniquely identifies an ORB in its address space. For example, the address space might be the server containing the ORB. The ID can be any String. The default value is a randomly generated number that is unique in the JVM of the ORB.

- **org.omg.CORBA.ORBListenEndpoints**

  Identifies the set of endpoints on which the ORB listens for requests. Each endpoint consists of a host name or IP address, and optionally a port. The value you specify is a string of the form `hostname:portnumber`, where the `:portnumber` component is optional. IPv6 addresses must be surrounded by brackets (for example, `[::1]:1020`). Specify multiple endpoints in a comma-separated list.

  Note: Some versions of the ORB support only the first endpoint in a multiple endpoint list.

  If this property is not set, the port number is set to 0 and the host address is retrieved by calling `InetAddress.getLocalHost().getHostAddress()`. If you specify only the host address, the port number is set to 0. If you want to set only the port number, you must also specify the host. You can specify the host name as the default host name of the ORB. The default host name is `localhost`.
- **org.omg.CORBA.ORBServerId**

  Assign the same value for this property to all ORBs contained in the same server. It is included in all IORs exported by the server. The integer value is in the range 0 - 2147483647).

This table shows the Java properties defined by Oracle Corporation that are now deprecated, and the IBM properties that have replaced them. These properties are not OMG standard properties, despite their names:

Oracle Corporation property	IBM property
com.sun.CORBA.ORBServerHost	com.ibm.CORBA.LocalHost
com.sun.CORBA.ORBServerPort	com.ibm.CORBA.ListenerPort
org.omg.CORBA.ORBInitialHost	com.ibm.CORBA.BootstrapHost
org.omg.CORBA.ORBInitialPort	com.ibm.CORBA.BootstrapPort
org.omg.CORBA.ORBInitialServices	com.ibm.CORBA.InitialReferencesURL

How the ORB works

This description tells you how the ORB works, by explaining what the ORB does transparently for the client. An important part of the work is performed by the server side of the ORB.

This section describes a basic, typical RMI-IIOP session in which a client accesses a remote object on a server. The access is made possible through an interface named Sample. The client calls a simple method provided through the interface. The method is called message(). The method returns a "Hello World" string. For further examples, see "Examples of client–server applications" on page 66.

The client side

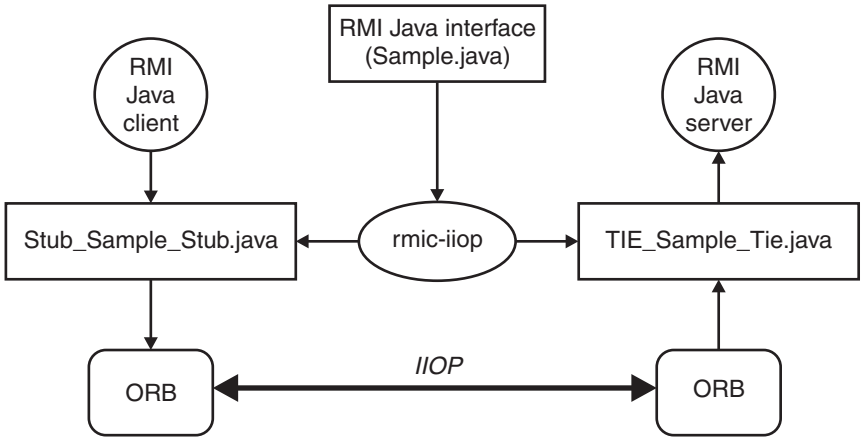
There are several steps to perform in order to enable an application client to use the ORB.

The subjects discussed here are:

Stub creation:

For any distributed application, the client must know what object it is going to contact, and which method of this object it must call. Because the ORB is a general framework, you must give it general information about the method that you want to call.

You provide the connection information by implementing a Java interface, for example Sample. The interface contains basic information about the methods that can be called in the remote object.



The client relies on the existence of a server containing an object that implements the Sample interface. You create a proxy object that is available on the client side for the client application to use. The proxy object is called a *stub*. The stub that acts as an interface between the client application and the ORB.

To create the stub, run the RMIC compiler on the Java interface:

```
rmic -iiop Sample
```

This command generates a file and object named `_Sample_Stub.class`.

The presence of a stub is not always mandatory for a client application to operate. When you use particular CORBA features such as the Dynamic Invocation Interface (DII), you do not require a stub. The reason is that the proxy code is implemented directly by the client application. You can also upload a stub from the server to which you are trying to connect. See the CORBA specification for further details.

ORB initialization:

In a stand-alone Java application, the client must create an instance of the ORB.

The ORB instance is created by calling the static method `init(...)`. For example:

```
ORB orb = ORB.init(args,props);
```

The parameters that are passed to the method are:
- A string array containing property-value pairs.
- A Java Properties object.

A similar method is used for an applet. The difference is that a Java Applet is passed instead of the string array.

The first step of ORB initialization is to process the ORB properties. The properties are found by searching in the following sequence:

1. First, check in the applet parameter, or application string array.
2. Check in the properties parameter, if the parameter exists.
3. Check in the system properties.
4. Check in any `orb.properties` file that is found in the `<user-home>` directory.
5. Check in any `orb.properties` file that is found in the `<java-home>/lib` directory.
6. Finally, use hardcoded default behavior.

Two important properties are **`ORBClass`** and **`ORBSingletonClass`**. These properties determine which ORB class is created and initialized, or "instantiated".

After the ORB is instantiated, it starts and initializes the TCP transport layer. If the **`ListenerPort`** property was set, the ORB also opens a server socket to listen for incoming requests. The **`ListenerPort`** property is used by a server-side ORB. At the end of the initialization performed by the init() method, the ORB is fully functional and ready to support the client application.

Obtaining the remote object:

Several methods exist by which the client can get a reference for the remote object.

Typically, this reference is a string, called an Interoperable Object Reference (IOR). For example:

```
IOR:000000000000001d524d493a5......
```

This reference contains all the information required to find the remote object. It also contains some details of the server settings to which the object belongs.

The client ORB does not have to understand the details of the IOR. The IOR is used as a reference to the remote object, like a key. However, when client and server are both using an IBM ORB, extra features are coded in the IOR. For example, the IBM ORB adds a proprietary field into the IOR, called **`IBM_PARTNER_VERSION`**. This field holds a value like the following example:

```
49424d0a 00000008 00000000 1400 0005
```

In the example:

- The first three bytes are the ASCII code for IBM
- The next byte is `0x0A`, which specifies that the following bytes provide information about the partner version.
- The next 4 bytes encode the length of the remaining data. In this example, the remaining data is 8 bytes long.
- The next 4 null bytes are reserved for future use.
- The next 2 bytes are for the Partner Version Major field. In this example, the value is `0x1400`, which means that release 1.4.0 of the ORB is being used.
- The final 2 bytes in this example have the value `0x0005` and represent the Minor field. This field is used to distinguish service refreshes within the same release. The service refreshes contain changes that affect compatibility with earlier versions.

The final step is called the “bootstrap process”. This step is where the client application tells the ORB where the remote object reference is located. The step is necessary for two reasons:

- The IOR is not visible to application-level ORB programmers.
- The client ORB does not know where to look for the IOR.

A typical example of the bootstrap process takes place when you use a naming service. First, the client calls the ORB method resolve_initial_references("NameService"). The method which returns a reference to the name server. The reference is in the form of a NamingContext object. The ORB then looks for a corresponding name server in the local system at the default port 2809. If no name server exists, or the name server cannot be found because it is listening on another port, the ORB returns an exception. The client application can specify a different host, a different port, or both, by using the **-ORBInitRef** and **-ORBInitPort** options.

Using the NamingContext and the name with which the Remote Object has been bound in the name service, the client can retrieve a reference to the remote object. The reference to the remote object that the client holds is always an instance of a Stub object; for example _Sample_Stub.

Using `ORB.resolve_initial_references()` causes much system activity. The ORB starts by creating a remote communication with the name server. This communication might include several requests and replies. Typically, the client ORB first checks whether a name server is listening. Next, the client ORB asks for the specified remote reference. In an application where performance is important, caching the remote reference is preferable to repetitive use of the naming service. However, because the naming service implementation is a transient type, the validity of the cached reference is limited to the time in which the naming service is running.

The IBM ORB implements an Interoperable Naming Service as described in the CORBA 2.3 specification. This service includes a new string format that can be passed as a parameter to the ORB methods string_to_object() and resolve_initial_references(). The methods are called with a string parameter that has a **corbaloc** (or **corbaname**) format. For example:

```
corbaloc:iiop:1.0@aserver.aworld.aorg:1050/AService
```

In this example, the client ORB uses GIOP 1.0 to send a request with a simple object key of **AService** to port 1050 at host `aserver.aworld.aorg`. There, the client ORB expects to find a server for the requested **Aservice**. The server replies by returning a reference to itself. You can then use this reference to look for the remote object.

This naming service is transient. It means that the validity of the contained references expires when the name service or the server for the remote object is stopped.

Remote method invocation:

The client holds a reference to the remote object that is an instance of the stub class. The next step is to call the method on that reference. The stub implements the Sample interface and therefore contains the message() method that the client has called.

First, the stub code determines whether the implementation of the remote object is located on the same ORB instance. If so, the object can be accessed without using the Internet.

If the implementation of the remote object is located on the same ORB instance, the performance improvement can be significant because a direct call to the object implementation is done. If no local servant can be found, the stub first asks the ORB to create a request by calling the _request() method, specifying the name of the method to call and whether a reply is expected or not.

The CORBA specification imposes an extra layer of indirection between the ORB code and the stub. This layer is commonly known as *delegation*. CORBA imposes the layer using an interface named Delegate. This interface specifies a portable API for ORB-vendor-specific implementation of the org.omg.CORBA.Object methods. Each stub contains a delegate object, to which all org.omg.CORBA.Object method invocations are forwarded. Using the delegate object means that a stub generated by the ORB from one vendor is able to work with the delegate from the ORB of another vendor.

When creating a request, the ORB first checks whether the **enableLocateRequest** property is set to `true`, in which case, a LocateRequest is created. The steps of creating this request are like the full Request case.

The ORB obtains the IOR of the remote object (the one that was retrieved by a naming service, for example) and passes the information that is contained in the IOR (Profile object) to the transport layer.

The transport layer uses the information that is in the IOR (IP address, port number, and object key) to create a connection if it does not exist. The ORB TCP/IP transport has an implementation of a table of cached connections for improving performances, because the creation of a new connection is a time-consuming process. The connection is not an open communication channel to the server host. It is only an object that has the potential to create and deliver a TCP/IP message to a location on the Internet. Typically, that involves the creation of a Java socket and a reader thread that is ready to intercept the server reply. The ORB.connect() method is called as part of this process.

When the ORB has the connection, it proceeds to create the Request message. The message contains the header and the body of the request. The CORBA 2.3 specification specifies the exact format. The header contains these items:

- Local IP address
- Local port
- Remote IP address
- Remote port
- Message size
- Version of the CORBA stream format
- Byte sequence convention
- Request types
- IDs

See “ORB problem determination” on page 196 for a detailed description and example.

The body of the request contains several service contexts and the name and parameters of the method invocation. Parameters are typically serialized.

A service context is some extra information that the ORB includes in the request or reply, to add several other functions. CORBA defines a few service contexts, such as the codebase and the codeset service contexts. The first is used for the callback feature which is described in the CORBA specification. The second context is used to specify the encoding of strings.

In the next step, the stub calls _invoke(). The effect is to run the delegate invoke() method. The ORB in this chain of events calls the send() method on the connection that writes the request to the socket buffer and then flushes it away. The delegate invoke() method waits for a reply to arrive. The reader thread that was spun during the connection creation gets the reply message, processes it, and returns the correct object.

The server side

In ORB terminology, a server is an application that makes one of its implemented objects available through an ORB instance.

The subjects discussed here are:

- "Servant implementation"
- "Tie generation"
- "Servant binding"
- "Processing a request" on page 81

Servant implementation:

The implementations of the remote object can either inherit from javax.rmi.PortableRemoteObject, or implement a remote interface and use the exportObject() method to register themselves as a servant object. In both cases, the servant has to implement the Sample interface. Here, the first case is described. From now, the servant is called SampleImpl.

Tie generation:

You must put an interfacing layer between the servant and the ORB code. In the old RMI (JRMP) naming convention, *skeleton* was the name given to the proxy that was used on the server side between ORB and the object implementation. In the RMI-IIOP convention, the proxy is called a Tie.

You generate the RMI-IIOP tie class at the same time as the stub, by calling the **rmic** compiler. These classes are generated from the compiled Java programming language classes that contain remote object implementations. For example, the command:

```
rmic -iiop SampleImpl
```

generates the stub `_Sample_Stub.class` and the tie `_Sample_Tie.class`.

Servant binding:

The steps required to bind the servant are described.

The server implementation is required to do the following tasks:

1. Create an ORB instance; that is, `ORB.init(...)`

2. Create a servant instance; that is, new `SampleImpl(...)`
3. Create a Tie instance from the servant instance; that is, `Util.getTie(...)`
4. Export the servant by binding it to a naming service

As described for the client side, you must create the ORB instance by calling the ORB static method init(...). The typical steps performed by the init(...) method are:

1. Retrieve properties
2. Get the system class loader
3. Load and instantiate the ORB class as specified in the ORBClass property
4. Initialize the ORB as determined by the properties

Next, the server must create an instance of the servant class SampleImpl.class. Something more than the creation of an instance of a class happens under the cover. Remember that the servant SampleImpl extends the PortableRemoteObject class, so the constructor of PortableRemoteObject is called. This constructor calls the static method exportObject(...) with the parameter that is the same servant instance that you try to instantiate. If the servant does not inherit from PortableRemoteObject, the application must call exportObject() directly.

The exportObject() method first tries to load an RMI-IIOP tie. The ORB implements a cache of classes of ties for improving performance. If a tie class is not already cached, the ORB loads a tie class for the servant. If it cannot find one, it goes up the inheritance tree, trying to load the parent class ties. The ORB stops if it finds a PortableRemoteObject class or the java.lang.Object, and returns a null value. Otherwise, it returns an instance of that tie from a hashtable that pairs a tie with its servant. If the ORB cannot find the tie, it assumes that an RMI (JRMP) skeleton might be present and calls the exportObject() method of the UnicastRemoteObject class. A null tie is registered in the cache and an exception is thrown. The servant is now ready to receive remote methods invocations. However, it is not yet reachable.

In the next step, the server code must find the tie itself (assuming the ORB has already got hold of the tie) to be able to export it to a naming service. To do that, the server passes the newly created instance of the servant into the static method javax.rmi.CORBA.Util.getTie(). This method, in turn, gets the tie that is in the hashtable that the ORB created. The tie contains the pair of tie-servant classes.

When in possession of the tie, the server must get hold of a reference for the naming service and bind the tie to it. As in the client side, the server calls the ORB method resolve_initial_references("NameService"). The server then creates a NameComponent, which is a directory tree object identifying the path and the name of the remote object reference in the naming service. The server binds the NameComponent together with the tie. The naming service then makes the IOR for the servant available to anyone requesting. During this process, the server code sends a LocateRequest to get hold of the naming server address. It also sends a Request that requires a rebind operation to the naming server.

Processing a request:

The server ORB uses a single listener thread, and a reader thread for each connection or client, to process an incoming message.

During the ORB initialization, a listener thread was created. The listener thread is listening on a default port (the next available port at the time the thread was created). You can specify the listener port by using the

com.ibm.CORBA.ListenerPort property. When a request comes in through that port, the listener thread first creates a connection with the client side. In this case, it is the TCP transport layer that takes care of the details of the connection. The ORB caches all the connections that it creates.

By using the connection, the listener thread creates a reader thread to process the incoming message. When dealing with multiple clients, the server ORB has a single listener thread and one reader thread for each connection or client.

The reader thread does not fully read the request message, but instead creates an input stream for the message to be piped into. Then, the reader thread picks up one of the worker threads in the implemented pool, or creates one if none is present. The work thread is given the task of reading the message. The worker thread reads all the fields in the message and dispatches them to the tie. The tie identifies any parameters, then calls the remote method.

The service contexts are then created and written to the response output stream with the return value. The reply is sent back with a similar mechanism, as described in the client side. Finally, the connection is removed from the reader thread which stops.

Additional features of the ORB

Portable object adapter, fragmentation, portable interceptors, and Interoperable Naming Service are described.

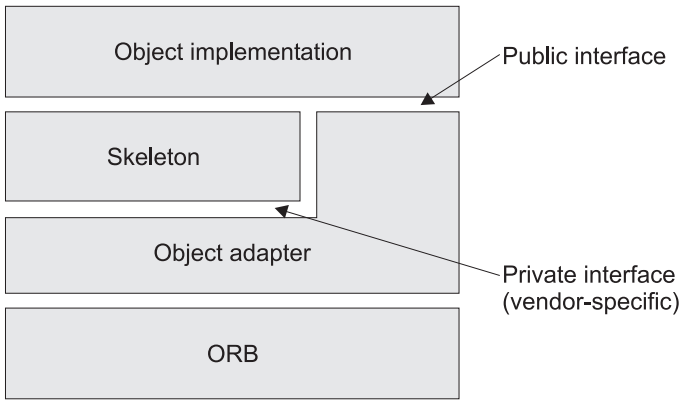
This section describes:

Portable object adapter

An object adapter is the primary way for an object to access ORB services such as object reference generation. A portable object adapter exports standard interfaces to the object.

The main responsibilities of an object adapter are:

- Generation and interpretation of object references.
- Enabling method calling.
- Object and implementation activation and deactivation.
- Mapping object references to the corresponding object implementations.



For CORBA 2.1 and earlier, all ORB vendors implemented an object adapter, which was known as the basic object adapter. A basic object adapter could not be specified with a standard CORBA IDL. Therefore, vendors implemented the adapters in many different ways. The result was that programmers were not able to write server implementations that were truly portable between different ORB products. A first attempt to define a standard object adapter interface was done in CORBA 2.1. With CORBA v.2.3, the OMG group released the final corrected version of a standard interface for the object adapter. This adapter is known as the Portable Object Adapter (POA).

Some of the main features of the POA specification are to:

- Allow programmers to construct object and server implementations that are portable between different ORB products.
- Provide support for persistent objects. The support enables objects to persist across several server lifetimes.
- Support transparent activation of objects.
- Associate policy information with objects.
- Allow multiple distinct instances of the POA to exist in one ORB.

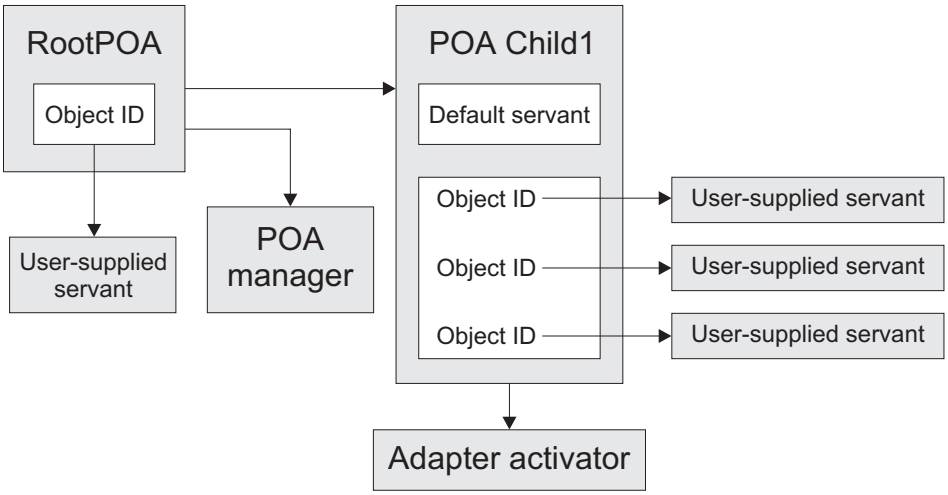
For more details of the POA, see the CORBA v.2.3 (formal/99-10-07) specification.

From IBM SDK for Java v1.4, the ORB supports both the POA specification and the proprietary basic object adapter that is already present in previous IBM ORB versions. By default, the RMI compiler, when used with the `-iiop` option, generates RMI-IIOP ties for servers. These ties are based on the basic object adapter. When a server implementation uses the POA interface, you must add the `-poa` option to the **`rmic`** compiler to generate the relevant ties.

To implement an object using the POA, the server application must obtain a POA object. When the server application calls the ORB method resolve_initial_reference("RootPOA"), the ORB returns the reference to the main POA object that contains default policies. For a list of all the POA policies, see the CORBA specification. You can create new POAs as child objects of the RootPOA. These child objects can contain different policies. This structure allows you to manage different sets of objects separately, and to partition the namespace of objects IDs.

Ultimately, a POA handles Object IDs and active servants. An active servant is a programming object that exists in memory. The servant is registered with the POA because one or more associated object identities was used. The ORB and POA cooperate to determine which servant starts the operation requested by the client.

By using the POA APIs, you can create a reference for the object, associate an object ID, and activate the servant for that object. A map of object IDs and active servants is stored inside the POA. A POA also provides a default servant that is used when no active servant has been registered. You can register a particular implementation of this default servant. You can also register a servant manager, which is an object for managing the association of an object ID with a particular servant.



The POA manager is an object that encapsulates the processing state of one or more POAs. You can control and change the state of all POAs by using operations on the POA manager.

The adapter activator is an object that an application developer uses to activate child POAs.

Fragmentation

The CORBA specification introduced the concept of fragmentation to handle the growing complexity and size of marshalled objects in GIOP messages. Graphs of objects are linearized and serialized inside a GIOP message under the IDL specification of valuetypes. Fragmentation specifies the way a message can be split into several smaller messages (fragments) and sent over the net.

The system administrator can set the ORB properties **FragmentSize** and **FragmentTimeout** to obtain best performance in the existing net traffic. As a general rule, the default value of 1024 bytes for the fragment size is a good trade-off in almost all conditions. The fragment timeout must not be set to too low a value, or time-outs might occur unnecessarily.

Portable interceptors

You can include “interceptor” code in the ORB processing flow. The CORBA 2.4.2 specification standardizes this code mechanism under the name “portable interceptor”.

CORBA implementations have mechanisms for users to insert their own code into the ORB processing flow. The code is inserted into the flow at “interception points”. The result is that the code, known as an interceptor, is called at particular stages during the processing of requests. It can directly inspect and even

manipulate requests. Because this message filtering mechanism is flexible and powerful, the OMG standardized interceptors in the CORBA 2.4.2 specification under the name “portable interceptors”.

The idea of a portable interceptor is to define a standard interface. The interface enables you to register and run application-independent code that, among other things, takes care of passing service contexts. These interfaces are stored in the package org.omg.PortableInterceptor.*. The implementation classes are in the com.ibm.rmi.pi.* package of the IBM ORB. All the interceptors implement the Interceptor interface.

Two classes of interceptors are defined:

Request interceptors
The ORB calls request interceptors on the client and the server side, during request mediation. Request interceptors manipulate service context information.

Interoperable Object Reference (IOR) interceptors
IOR interceptors are called when new object references are created. The reason is that service-specific data, in the form of tagged components, can be added to the newly created IOR.

Interceptors must register with the ORB for the interception points where they are to run.

Five interception points are available on the client side:

- send_request (sending request)
- send_poll (sending request)
- receive_reply (receiving reply)
- receive_exception (receiving reply)
- receive_other (receiving reply)

Five interception points are available on the server side:

- receive_request_service_contexts (receiving request)
- receive_request (receiving request)
- send_reply (sending reply)
- send_exception (sending reply)
- send_other (sending reply)

The only interception point for IOR interceptors is establish_component(). The ORB calls this interception point on all its registered IOR interceptors when it is assembling the set of components that is to be included in the IOP profiles for a new object reference.

A simple interceptor is shown in the following example:

```
public class MyInterceptor extends org.omg.CORBA.LocalObject
 implements ClientRequestInterceptor, ServerRequestInterceptor
{
 public String name() {
  return "MyInterceptor";
 }

 public void destroy() {}

 // ClientRequestInterceptor operations
```

```
 public void send_request(ClientRequestInfo ri) {
  logger(ri, "send_request");
 }

 public void send_poll(ClientRequestInfo ri) {
  logger(ri, "send_poll");
 }

 public void receive_reply(ClientRequestInfo ri) {
  logger(ri, "receive_reply");
 }

 public void receive_exception(ClientRequestInfo ri) {
  logger(ri, "receive_exception");
 }

 public void receive_other(ClientRequestInfo ri) {
  logger(ri, "receive_other");
 }

 // Server interceptor methods
 public void receive_request_service_contexts(ServerRequestInfo ri) {
  logger(ri, "receive_request_service_contexts");
 }

 public void receive_request(ServerRequestInfo ri) {
  logger(ri, "receive_request");
 }

 public void send_reply(ServerRequestInfo ri) {
  logger(ri, "send_reply");
 }

 public void send_exception(ServerRequestInfo ri) {
  logger(ri, "send_exception");
 }

 public void send_other(ServerRequestInfo ri) {
  logger(ri, "send_other");
 }

 // Trivial Logger
 public void logger(RequestInfo ri, String point) {
  System.out.println("Request ID:" + ri.request_id()
   + " at " + name() + "." + point);
 }
}
```

The interceptor class extends org.omg.CORBA.LocalObject. The extension ensures that an instance of this class does not get marshaled, because an interceptor instance is tied to the ORB with which it is registered. This example interceptor prints out a message at every interception point.

You cannot register an interceptor with an ORB instance after it has been created. The reason is that interceptors are a means for ORB services to interact with ORB processing. Therefore, by the time the init() method call on the ORB class returns an ORB instance, the interceptors must already be registered. Otherwise, the interceptors are not part of the ORB processing flow.

You register an interceptor by using an ORB initializer. First, you create a class that implements the **ORBInitializer** interface. This class is called by the ORB during its initialization.

```
public class MyInterceptorORBInitializer extends LocalObject
 implements ORBInitializer
{
 public static MyInterceptor interceptor;

 public String name() {
  return "";
 }

 public void pre_init(ORBInitInfo info) {
  try {
   interceptor = new MyInterceptor();
   info.add_client_request_interceptor(interceptor);
   info.add_server_request_interceptor(interceptor);
  } catch (Exception ex) {}
 }

 public void post_init(ORBInitInfo info) {}

}
```

Then, in the server implementation, add the following code:

```
Properties p = new Properties();
p.put("org.omg.PortableInterceptor.ORBInitializerClass.pi.
               MyInterceptorORBInitializer", "");
...
orb = ORB.init((String[])null, p);
```

During the ORB initialization, the ORB run time code obtains the ORB properties with names that begin with org.omg.PortableInterceptor.ORBInitializerClass. The remaining portion of the name is extracted, and the corresponding class is instantiated. Then, the pre_init() and post_init() methods are called on the initializer object.

Interoperable Naming Service (INS)

The CORBA "CosNaming" Service follows the Object Management Group (OMG) Interoperable Naming Service specification (INS, CORBA 2.3 specification). CosNaming stands for Common Object Services Naming.

The name service maps names to CORBA object references. Object references are stored in the namespace by name and each object reference-name pair is called a name *binding*. Name bindings can be organized under *naming contexts*. Naming contexts are themselves name bindings, and serve the same organizational function as a file system subdirectory does. All bindings are stored under the initial naming context. The initial naming context is the only persistent binding in the namespace.

This implementation includes string formats that can be passed as a parameter to the ORB methods string_to_object() and resolve_initial_references(). The formats are `corbaname` and `corbaloc`.

`Corbaloc` URIs allow you to specify object references that can be contacted by IIOP or found through ORB::resolve_initial_references(). This format is easier to manipulate than IOR. To specify an IIOP object reference, use a URI of the form:

```
corbaloc:iiop:<host>:<port>/<object key>
```

Note: See the CORBA 2.4.2 specification for the full syntax of this format.

For example, the following `corbaloc` URI specifies an object with key **`MyObjectKey`** that is in a process that is running on `myHost.myOrg.com`, listening on port 2809:

```
corbaloc:iiop:myHost.myOrg.com:2809/MyObjectKey
```

Corbaname URIs cause the string_to_object() method to look up a name in a CORBA naming service. The URIs are an extension of the corbaloc syntax:

```
corbaname:<corbaloc location>/<object key>#<stringified name>
```

Note: See the CORBA 2.4.2 specification for the full syntax of this format.

An example corbaname URI is:

```
corbaname::myOrg.com:2050#Personal/schedule
```

In this example, the portion of the reference up to the number sign character "#" is the URL that returns the root naming context. The second part of the example, after the number sign character "#", is the argument that is used to resolve the object on the NamingContext.

The INS specified two standard command-line arguments that provide a portable way of configuring ORB::resolve_initial_references():

- **-ORBInitRef** takes an argument of the form <ObjectId>=<ObjectURI>. For example, you can use the following command-line arguments:

  ```
  -ORBInitRef NameService=corbaname::myhost.example.com
  ```

  In this example, resolve_initial_references("NameService") returns a reference to the object with key NameService available on myhost.example.com, port 2809.
- **-ORBDefaultInitRef** provides a prefix string that is used to resolve otherwise unknown names. When resolve_initial_references() cannot resolve a name that has been configured with **-ORBInitRef**, it constructs a string that consists of the default prefix, a "/" character, and the name requested. The string is then supplied to string_to_object(). For example, with a command line of:

  ```
  -ORBDefaultInitRef corbaloc::myhost.example.com
  ```

  a call to resolve_initial_references("MyService") returns the object reference that is denoted by corbaloc::myhost.example.com/MyService.

The Java Native Interface (JNI)

This description of the Java Native Interface (JNI) provides background information to help you diagnose problems with JNI operation.

The specification for the Java Native Interface (JNI) is maintained by Oracle Corporation. IBM recommends that you read the JNI specification. Go to http://www.oracle.com/technetwork/java/index.html and search the site for JNI. Oracle Corporation maintain a combined programming guide and specification at http://java.sun.com/docs/books/jni/.

This section gives additional information to help you with JNI operation and design.

The topics that are discussed in this section are:

Overview of JNI

From the viewpoint of a JVM, there are two types of code: "Java" and "native". The Java Native Interface (JNI) establishes a well-defined and platform-independent interface between the two.

Native code can be used together with Java in two distinct ways: as "native methods" in a running JVM and as the code that creates a JVM using the "Invocation API". This section describes the difference.

Native methods

Java native methods are declared in Java, implemented in another language (such as C or C++), and loaded by the JVM as necessary. To use native methods, you must:

1. **Declare** the native method in your Java code.

   When the javac compiler encounters a native method declaration in Java source code, it records the name and parameters for the method. Because the Java source code contains no implementation, the compiler marks the method as "native". The JVM can then resolve the method correctly when it is called.
2. **Implement** the native method.

   Native methods are implemented as external entry points in a loadable binary library. The contents of a native library are platform-specific. The JNI provides a way for the JVM to use any native methods in a platform-independent way.

   The JVM performs calls to native methods. When the JVM is in a native method, JNI provides a way to "call back" to the JVM.
3. **Load** the native method code for the VM to use.

   As well as declaring the native method, you must find and load the native library that contains the method at run time.

   Two Java interfaces load native libraries:

   - java.lang.System.load()
   - java.lang.System.loadLibrary()

   Typically, a class that declares native methods loads the native library in its static initializer.

Invocation API

Creating a JVM involves native code. The aspect of the JNI used for this purpose is called the JNI Invocation API. To use the Invocation API, you bind to an implementation-specific shared library, either statically or dynamically, and call the JNI_* functions it exports.

The JNI specification and implementation

The JNI specification is vague on selected implementation details. It provides a reusable framework for simple and extensible C and C++ native interfaces. The JNI model is also the basis for the JVMTI specification.

The Oracle Corporation trademark specification and the Java Compatibility Kit (JCK) ensure compliance to the specification but not to the implementation. Native

code must conform to the specification and not to the implementation. Code written against unspecified behavior is prone to portability and forward compatibility problems.

The JNI and the Garbage Collector

This description explains how the JNI implementation ensures that objects can be reached by the Garbage Collector (GC).

For general information about the IBM GC, see “Memory management” on page 23.

To collect unreachable objects, the GC must know when Java objects are referenced by native code. The JNI implementation uses "root sets" to ensure that objects can be reached. A root set is a set of direct, typically relocatable, object references that are traceable by the GC.

There are several types of root set. The union of all root sets provides the starting set of objects for a GC mark phase. Beginning with this starting set, the GC traverses the entire object reference graph. Anything that remains unmarked is unreachable garbage. (This description is an over-simplification when reachability and weak references are considered. See “Detailed description of global garbage collection” on page 29 and the JVM specification.)

Overview of JNI object references

The implementation details of how the GC finds a JNI object reference are not detailed in the JNI specification. Instead, the JNI specifies a required behavior that is both reliable and predictable.

Local and global references

Local references are scoped to their creating stack frame and thread, and automatically deleted when their creating stack frame returns. Global references allow native code to promote a local reference into a form usable by native code in any thread attached to the JVM.

Global references and memory leaks

Global references are not automatically deleted, so the programmer must handle the memory management. Every global reference establishes a root for the referent and makes its entire subtree reachable. Therefore, every global reference created must be freed to prevent memory leaks.

Leaks in global references eventually lead to an out-of-memory exception. These errors can be difficult to solve, especially if you do not perform JNI exception handling. See “Handling exceptions” on page 96.

To provide JNI global reference capabilities and also provide some automatic garbage collection of the referents, the JNI provides two functions:

- NewWeakGlobalRef
- DeleteWeakGlobalRef

These functions provide JNI access to weak references.

Local references and memory leaks

The automatic garbage collection of local references that are no longer in scope prevents memory leaks in most situations. This automatic garbage collection occurs when a native thread returns to Java (native methods) or detaches from the JVM (Invocation API). Local reference memory leaks are possible if automatic garbage collection does not occur. A memory leak might occur if a native method does not return to the JVM, or if a program that uses the Invocation API does not detach from the JVM.

Consider the code in the following example, where native code creates new local references in a loop:

```
while ( <condition> )
{
   jobject myObj = (*env)->NewObject( env, clz, mid, NULL );

   if ( NULL != myObj )
   {
      /* we know myObj is a valid local ref, so use it */
      jclass myClazz = (*env)->GetObjectClass(env, myObj);

      /* uses of myObj and myClazz, etc. but no new local refs */

      /* Without the following calls, we would leak */
      (*env)->DeleteLocalRef( env, myObj );
      (*env)->DeleteLocalRef( env, myClazz );
   }

} /* end while */
```

Although new local references overwrite the `myObj` and `myClazz` variables inside the loop, every local reference is kept in the root set. These references must be explicitly removed by the DeleteLocalRef call. Without the DeleteLocalRef calls, the local references are leaked until the thread returned to Java or detached from the JVM.

JNI weak global references

Weak global references are a special type of global reference. They can be used in any thread and can be used between native function calls, but do not act as GC roots. The GC disposes of an object that is referred to by a weak global reference at any time if the object does not have a strong reference elsewhere.

You must use weak global references with caution. If the object referred to by a weak global reference is garbage collected, the reference becomes a null reference. A null reference can only safely be used with a subset of JNI functions. To test if a weak global reference has been collected, use the IsSameObject JNI function to compare the weak global reference to the null value.

It is not safe to call most JNI functions with a weak global reference, even if you have tested that the reference is not null, because the weak global reference could become a null reference after it has been tested or even during the JNI function. Instead, a weak global reference should always be promoted to a strong reference before it is used. You can promote a weak global reference using the NewLocalRef or NewGlobalRef JNI functions.

Weak global references use memory and must be freed with the DeleteWeakGlobalRef JNI function when it is no longer needed. Failure to free weak global references causes a slow memory leak, eventually leading to out-of-memory exceptions.

For information and warnings about the use of JNI global weak references, see the JNI specification.

JNI reference management

There are a set of platform-independent rules for JNI reference management

These rules are:

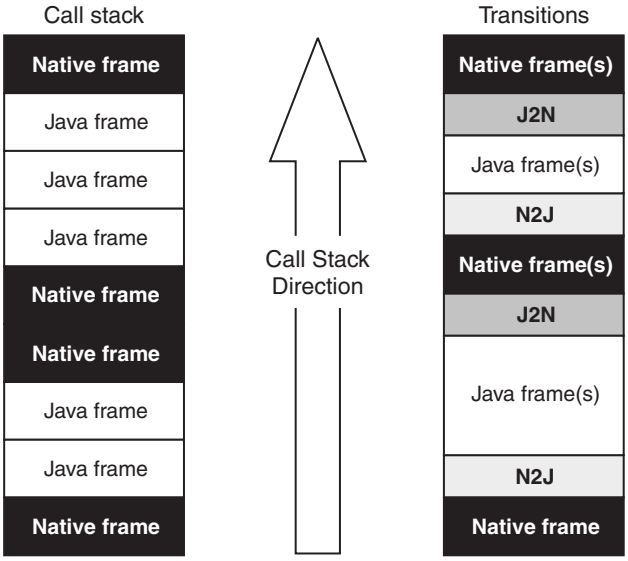
1. JNI references are valid only in threads attached to a JVM.
2. A valid JNI local reference in native code must be obtained:
   a. As a parameter to the native code
   b. As the return value from calling a JNI function
3. A valid JNI global reference must be obtained from another valid JNI reference (global or local) by calling NewGlobalRef or NewWeakGlobalRef.
4. The null value reference is always valid, and can be used in place of any JNI reference (global or local).
5. JNI local references are valid only in the thread that creates them and remain valid only while their creating frame remains on the stack.

Note:

1. Overwriting a local or global reference in native storage with a null value does not remove the reference from the root set. Use the appropriate Delete*Ref JNI function to remove references from root sets.
2. Many JNI functions (such as FindClass and NewObject) return a null value if there is an exception pending. Comparing the returned value to the null value for these calls is semantically equivalent to calling the JNI ExceptionCheck function. See the JNI specification for more details.
3. A JNI local reference must never be used after its creating frame returns, regardless of the circumstances. It is dangerous to store a JNI local reference in any process static storage.

JNI transitions

To understand JNI local reference management and the GC, you must understand the context of a running thread attached to the JVM. Every thread has a runtime stack that includes a frame for each method call. From a GC perspective, every stack establishes a thread-specific "root set" including the union of all JNI local references in the stack.



Each method call in a running VM adds (pushes) a frame onto the stack, just as every return removes (pops) a frame. Each call point in a running stack can be characterized as one of the following types:

- Java to Java (J2J)
- Native to Native (N2N)
- Java to Native (J2N)
- Native to Java (N2J)

You can only perform an N2J transition in a thread that meets the following conditions:

- The process containing the thread must contain a JVM started using the JNI Invocation API.
- The thread must be "attached" to the JVM.
- The thread must pass at least one valid local or global object reference to JNI.

J2J and N2N transitions:

Because object references do not change form as part of J2J or N2N transitions, J2J and N2N transitions do not affect JNI local reference management.

Any section of N2N code that obtains many local references without promptly returning to Java can needlessly stress the local reference capacity of a thread. This problem can be avoided if local references are managed explicitly by the native method programmer.

N2J transitions:

For native code to call Java code (N2J) in the current thread, the thread must first be attached to the JVM in the current process.

Every N2J call that passes object references must have obtained them using JNI, therefore they are either valid local or global JNI refs. Any object references returned from the call are JNI local references.

J2N calls:

The JVM must ensure that objects passed as parameters from Java to the native method and any new objects created by the native code remain reachable by the GC. To handle the GC requirements, the JVM allocates a small region of specialized storage called a *local reference root set*.

A local reference root set is created when:

- A thread is first attached to the JVM (the outermost root set of the thread).
- Each J2N transition occurs.

The JVM initializes the root set created for a J2N transition with:

- A local reference to the caller's object or class.
- A local reference to each object passed as a parameter to the native method.

New local references created in native code are added to this J2N root set, unless you create a new local frame using the PushLocalFrame JNI function.

The default root set is large enough to contain 16 local references per J2N transition. The **-Xcheck:jni** command-line option causes the JVM to monitor JNI usage. When **-Xcheck:jni** is used, the JVM writes a warning message when more than 16 local references are required at run time. If you receive this warning message, use one of the following JNI functions to manage local references more explicitly:

- NewLocalRef
- DeleteLocalRef
- PushLocalFrame
- PopLocalFrame
- EnsureLocalCapacity

J2N returns:

When native code returns to Java, the associated JNI local reference "root set", created by the J2N call, is released.

If the JNI local reference was the only reference to an object, the object is no longer reachable and can be considered for garbage collection. Garbage collection is triggered automatically by this condition, which simplifies memory management for the JNI programmer.

Copying and pinning

The GC might, at any time, decide it needs to compact the garbage-collected heap. Compaction involves physically moving objects from one address to another. These objects might be referred to by a JNI local or global reference. To allow compaction to occur safely, JNI references are not direct pointers to the heap. At least one level of indirection isolates the native code from object movement.

If a native method needs to obtain direct addressability to the inside of an object, the situation is more complicated. The requirement to directly address, or pin, the heap is typical where there is a need for fast, shared access to large primitive arrays. An example might include a screen buffer. In these cases a JNI critical section can be used, which imposes additional requirements on the programmer, as specified in the JNI description for these functions. See the JNI specification for details.

- GetPrimitiveArrayCritical returns the direct heap address of a Java array, disabling garbage collection until the corresponding ReleasePrimitiveArrayCritical is called.
- GetStringCritical returns the direct heap address of a java.lang.String instance, disabling garbage collection until ReleaseStringCritical is called.

All other Get<*PrimitiveType*>ArrayElements interfaces return a copy that is unaffected by compaction.

When using the Balanced Garbage Collection Policy, the *Critical forms of the calls might not return a direct pointer into the heap, which is reflected in the isCopy flag. This behavior is due to an internal representation of larger arrays, where data might not be sequential. Typically, an array with storage that is less than 1/1000th of the heap, is returned as a direct pointer.

Using the isCopy flag

The JNI Get<*Type*> functions specify a pass-by-reference output parameter (jboolean *isCopy) that allows the caller to determine whether a given JNI call is returning the address of a copy or the address of the pinned object in the heap.

The Get<*Type*> and Release<*Type*> functions come in pairs:

- GetStringChars and ReleaseStringChars
- GetStringCritical and ReleaseStringCritical
- GetStringUTFChars and ReleaseStringUTFChars
- Get<PrimitiveType>ArrayElements and Release<PrimitiveType>ArrayElements
- GetPrimitiveArrayCritical and ReleasePrimitiveArrayCritical

If you pass a non-null address as the isCopy parameter, the JNI function sets the jboolean value at that address to JNI_TRUE if the address returned is the address of a copy of the array elements and JNI_FALSE if the address points directly into the pinned object in the heap.

Except for the critical functions, the IBM JVM always returns a copy. Copying eases the burden on the GC, because pinned objects cannot be compacted and complicate defragmentation.

To avoid leaks, you must:

- Manage the copy memory yourself using the Get<*Type*>Region and Set<*Type*>Region functions.
- Ensure that you free copies made by a Get<*Type*> function by calling the corresponding Release<*Type*> function when the copy is no longer needed.

Using the mode flag

When you call Release<*Type*>ArrayElements, the last parameter is a mode flag. The mode flag is used to avoid unnecessary copying to the Java heap when working with a copied array. The mode flag is ignored if you are working with an array that has been pinned.

You must call Release<*Type*> once for every Get<*Type*> call, regardless of the value of the isCopy parameter. This step is necessary because calling Release<*Type*> deletes JNI local references that might otherwise prevent garbage collection.

The possible settings of the mode flag are:

0 Update the data on the Java heap. Free the space used by the copy.

JNI_COMMIT
Update the data on the Java heap. **Do not** free the space used by the copy.

JNI_ABORT
Do not update the data on the Java heap. Free the space used by the copy.

The '0' mode flag is the safest choice for the Release<*Type*> call. Whether the copy of the data was changed or not, the heap is updated with the copy, and there are no leaks.

To avoid having to copy back an unchanged copy, use the JNI_ABORT mode value. If you alter the returned array, check the isCopy flag before using the JNI_ABORT mode value to "roll back" changes. This step is necessary because a pinning JVM leaves the heap in a different state than a copying JVM.

A generic way to use the isCopy and mode flags

Here is a generic way to use the isCopy and mode flags. It works with all JVMs and ensures that changes are committed and leaks do not occur.

To use the flags in a generic way, ensure that you:

- Do not use the isCopy flag. Pass in null or 0.
- Always set the mode flag to zero.

A complicated use of these flags is necessary only for optimization. If you use the generic way, you must still consider synchronization. See "Synchronization."

Handling exceptions

Exceptions give you a way to handle errors in your application. Java has a clear and consistent strategy for the handling of exceptions, but C/C++ code does not. Therefore, the Java JNI does not throw an exception when it detects a fault. The JNI does not know how, or even if, the native code of an application can handle it.

The JNI specification requires exceptions to be deferred; it is the responsibility of the native code to check whether an exception has occurred. A set of JNI APIs are provided for this purpose. A JNI function with a return code always sets an error if an exception is pending. You do not need to check for exceptions if a JNI function returns "success", but you must check for an exception in an error case. If you do not check, the next time you go through the JNI, the JNI code detects a pending exception and throws it. An exception can be difficult to debug if it is thrown later and, possibly, at a different point in the code from the point at which it was created.

Note: The JNI ExceptionCheck function is a more optimal way of doing exception checks than the ExceptionOccurred call, because the ExceptionOccurred call has to create a local reference.

Synchronization

When you get array elements through a Get<*Type*>ArrayElements call, you must think about synchronization.

Whether the data is pinned or not, two entities are involved in accessing the data:

- The Java code in which the data entity is declared and used
- The native code that accesses the data through the JNI

These two entities are probably separate threads, in which case contention occurs.

Consider the following scenario in a copying JNI implementation:

1. A Java program creates a large array and partially fills it with data.
2. The Java program calls native write function to write the data to a socket.
3. The JNI native that implements `write()` calls GetByteArrayElements.
4. GetByteArrayElements copies the contents of the array into a buffer, and returns it to the native.
5. The JNI native starts writing a region from the buffer to the socket.
6. While the thread is busy writing, another thread (Java or native) runs and copies more data into the array (outside the region that is being written).
7. The JNI native completes writing the region to the socket.
8. The JNI native calls ReleaseByteArrayElements with mode 0, to indicate that it has completed its operation with the array.
9. The VM, seeing mode 0, copies back the whole contents of the buffer to the array, and overwrites the data that was written by the second thread.

In this particular scenario, the code works with a pinning JVM. Because each thread writes only its own bit of the data and the mode flag is ignored, no contention occurs. This scenario is another example of how code that is not written strictly to specification works with one JVM implementation and not with another. Although this scenario involves an array elements copy, pinned data can also be corrupted when two threads access it at the same time.

Be careful about how you synchronize access to array elements. You can use the JNI interfaces to access regions of Java arrays and strings to reduce problems in this type of interaction. In the scenario, the thread that is writing the data writes into its own region. The thread that is reading the data reads only its own region. This method works with every JNI implementation.

Debugging the JNI

If you think you have a JNI problem, there are checks you can run to help you diagnose the JNI transitions.

Errors in JNI code can occur in several ways:

- The program crashes during execution of a native method (most common).
- The program crashes some time after returning from the native method, often during GC (not so common).
- Bad JNI code causes deadlocks shortly after returning from a native method (occasional).

If you think that you have a problem with the interaction between user-written native code and the JVM (that is, a JNI problem), you can run checks that help you diagnose the JNI transitions. To run these checks, specify the **-Xcheck:jni** option when you start the JVM.

The **-Xcheck:jni** option activates a set of wrapper functions around the JNI functions. The wrapper functions perform checks on the incoming parameters. These checks include:

- Whether the call and the call that initialized JNI are on the same thread.
- Whether the object parameters are valid objects.
- Whether local or global references refer to valid objects.
- Whether the type of a field matches the `Get<Type>Field` or `Set<Type>Field` call.

- Whether static and nonstatic field IDs are valid.
- Whether strings are valid and non-null.
- Whether array elements are non-null.
- The types on array elements.

Output from **-Xcheck:jni** is displayed on the standard error stream, and looks like:

```
JVMJNCK059W: JNI warning in FindClass: argument #2 is a malformed identifier ("invalid.name")
JVMJNCK090W: Warning detected in com/ibm/examples/JNIExample.nativeMethod() [Ljava/lang/String];
```

The first line indicates:

- The error level (error, warning, or advice).
- The JNI API in which the error was detected.
- An explanation of the problem.

The last line indicates the native method that was being executed when the error was detected.

You can specify additional suboptions by using **-Xcheck:jni:<suboption>[,<...>]**. Useful suboptions are:

all
Check application and system classes.

verbose
Trace certain JNI functions and activities.

trace
Trace all JNI functions.

nobounds
Do not perform bounds checking on strings and arrays.

nonfatal
Do not exit when errors are detected.

nowarn
Do not display warnings.

noadvice
Do not display advice.

novalist
Do not check for va_list reuse (see the note at the end of this section).

pedantic
Perform more thorough, but slower checks.

valist
Check for va_list reuse (see the note at the end of the section).

help
Print help information.

The **-Xcheck:jni** option might reduce performance because it is thorough when it validates the supplied parameters.

Note:

On some platforms, reusing a va_list in a second JNI call (for example, when calling CallStaticVoidMethod() twice with the same arguments) causes the va_list

to be corrupted and the second call to fail. To ensure that the va_list is not corrupted, use the standard C macro va_copy() in the first call. By default, **-Xcheck:jni** ensures that va_lists are not being reused. Use the **novalist** suboption to disable this check only if your platform allows reusing va_list without va_copy. z/OS platforms allow va_list reuse, and by default **-Xcheck:jni:novalist** is used. To enable va_list reuse checking, use the **-Xcheck:jni:valist** option.

JNI checklist

There are a number of items that you must remember when using the JNI.

The following table shows the JNI checklist:

Remember	Outcome of nonadherence
Local references cannot be saved in global variables.	Random crashes (depending on what you pick up in the overwritten object space) happen at random intervals.
Ensure that every global reference created has a path that deletes that global reference.	Memory leak. It might throw a native exception if the global reference storage overflows. It can be difficult to isolate.
Always check for exceptions (or return codes) on return from a JNI function. Always handle a deferred exception immediately you detect it.	Unexplained exceptions or undefined behavior. Might crash the JVM.
Ensure that array and string elements are always freed.	A small memory leak. It might fragment the heap and cause other problems to occur first.
Ensure that you use the isCopy and mode flags correctly. See “A generic way to use the isCopy and mode flags” on page 96.	Memory leaks, heap fragmentation, or both.
When you update a Java object in native code, ensure synchronization of access.	Memory corruption.

Chapter 3. Planning

Information that you should know when planning to use or migrate the product, such as supported environments and version compatibility.

Migrating from earlier IBM SDK or JREs

Important information to consider before upgrading from earlier versions of the IBM SDK and JRE.

The IBM SDK for z/OS, V7 contains many new features and functions, which might present planning considerations. For an overview, read the "What's new" on page 6 section.

If you are migrating from IBM SDK Java Technology Edition V6, read the following points:

- The default garbage collection policy in force is now the Generational Concurrent garbage collector. For an overview, see "Generational Concurrent Garbage Collector" on page 37.
- 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 "How the JIT compiler optimizes code" on page 58.
- Shared class caches are now persistent by default on the AIX operating system. For more information, see "-Xshareclasses" on page 439.
- Compressed references are now the default on all platforms except z/OS when the value of the **-Xmx** option is less than or equal to 25 GB. For more information about the use of compressed references, see "Compressed references" on page 27.
- Verbose garbage collection logging is redesigned. See "Verbose garbage collection logging" on page 334.
- The JRIO component available in earlier versions of the IBM SDK for Java is deprecated and will be removed in future releases. As an alternative, use the record I/O facilities that are provided in the JZOS component.
- From service refresh 4: On certain platforms and processors, the JVM now starts with large pages enabled by default for both the JIT codecache and the objectheap, instead of the default operating system page size. For more information, see "-Xlp" on page 435.
- From service refresh 4: The SDK uses the Oracle implementation of the java.util.* package, including all classes within the package. Earlier releases of the SDK used customized versions of the Apache Harmony class libraries. This change establishes a common implementation point for the java.util.* package, enabling consistent performance and behavior characteristics across Java implementations.
- For additional industry compatibility information, see "Version compatibility" on page 102.
- For additional deprecated API information, see Oracle's Java 7 Deprecated API List: http://download.oracle.com/javase/7/docs/api/deprecated-list.html

Note: If you are migrating from IBM SDK Java Technology Edition V6 (J9 VM2.6), only a subset of this list applies because VM version 2.6 is also provided with the IBM SDK for z/OS, V7.

If you are migrating from a release before IBM SDK Java Technology Edition V6, read the topic entitled "Migrating from earlier IBM SDK or JREs" in the appropriate platform user guide for Java V6. These user guides can be found at the following URL: http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/index.jsp.

Version compatibility

In general, any application that ran with a previous version of the SDK should run correctly with the IBM SDK for z/OS, V7. Classes that are compiled with this release are not guaranteed to work on previous releases.

For information about compatibility issues between releases, see the Oracle Web site at:

http://www.oracle.com/technetwork/java/javase/compatibility-417013.html

Supported environments

The IBM SDK for z/OS, V7 is supported on certain hardware platforms and operating systems, and is tested on specific virtualization environments.

IBM SDK for z/OS

The z/OS 31-bit and 64-bit SDKs run on the following System z® hardware:

- zEnterprise EC12
- z196
- z10™
- z9-109
- z990
- z900
- z800
- z114

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 1. z/OS environments tested

Operating system	31-bit SDK	64-bit SDK
z/OS 1.10	GA	GA
z/OS 1.11	GA	GA
z/OS 1.12	GA	GA
z/OS 1.13	SR1	SR1

Virtualization software

For information about the virtualization software tested, see “Support for virtualization software” on page 479.

Chapter 4. Installing and configuring the SDK

See the z/OS Web site for instructions about ordering, downloading, installing, and verifying the SDK.

http://www.ibm.com/systems/z/os/zos/tools/java/

Working with BPXPRM settings

Some of the parameters in PARMLIB member **BPXPRMxx** might affect successful Java operation by imposing limits on resources that are required.

The parameters described here do not cover the ones required for Class data sharing. See "Considerations and limitations of using class data sharing" on page 166 for the parameters required for Class data sharing.

To see the current **BPXPRMxx** settings, enter the z/OS operator command `D OMVS,O`. To show the highwater usage for some of the limits, enter the command `D OMVS,L`. If you configure the **BPXPRMxx LIMMSG** parameter to activate the support, **BPXInnnI** messages are reported when the usage approaches and reaches the limits. You can use the SETOMVS command to change the settings without requiring an IPL.

Other products might impose their own requirements, but for Java the important parameters and their suggested minimum values are:

Table 2. BPXPRM settings

Parameter	Value
MAXPROCSYS	900
MAXPROCUSER	512
MAXUIDS	500
MAXTHREADS	10 000
MAXTHREADTASKS	5 000
MAXASSIZE	2 147 483 647
MAXCPUTIME	2 147 483 647
MAXMMAPAREA	40 960
IPCSEMNIDS	500
IPCSEMNSEMS	1 000
SHRLIBRGNSIZE	67 108 864
SHRLIBMAXPAGES	4 096

The lower of **MAXTHREADS** and **MAXTHREADTASKS** limits the number of threads that can be created by a Java process.

MAXMMAPAREA limits the number of 4K pages that are available for memory-mapped jar files through the environment variable **JAVA_MMAP_MAXSIZE**.

SHRLIBRGNSIZE controls how much storage is reserved in each address space for mapping shared DLLs that have the +l extended attribute set. If this storage space

is exceeded, DLLs are loaded into the address space instead of using a single copy of z/OS UNIX System Services storage that is shared between the address spaces. Some of the Java SDK DLLs have the +l extended attribute set. The z/OS command **D OMVS,L** shows the **SHRLIBRGNSIZE** size and peak usage. If this size is set to a much higher value than is needed, Java might have problems acquiring native (z/OS 31-bit) storage. These problems can cause a z/OS abend, such as 878-10, or a Java OutOfMemoryError.

SHRLIBMAXPAGES is only available in z/OS 1.7 and earlier releases. This parameter is like the **SHRLIBRGNSIZE** parameter, except that it is a number of 4K pages. The parameter only applies to DLLs that have the .so suffix, but without the +l extended attribute. This feature requires Extended System Queue Area (ESQA), therefore must be used carefully.

For further information about the use of these parameters, see:

- z/OS MVS™ Initialization and Tuning Reference (SA22-7592)
- z/OS Unix System Services Planning Guide (GA22-7800)

Setting the region size

Java requires a suitable z/OS region size to operate successfully. It is suggested that you do not restrict the region size, but allow Java to use what is necessary. Restricting the region size might cause failures with storage-related error messages or abends such as 878-10.

The region size might be affected by the following factors:

- **JCL REGION** parameter
- **BPXPRMxx MAXASSIZE** parameter
- RACF OMVS segment **ASSIZEMAX** parameter
- IEFUSI

You might want to exclude OMVS from using the IEFUSI exit by setting **SUBSYS(OMVS,NOEXITS)** in PARMLIB member SMFPRMxx.

For further information, see the documentation about the host product under which Java runs.

Setting MEMLIMIT

z/OS uses region sizes to determine the amount of storage available to running programs. For the 64-bit product, set the **MEMLIMIT** parameter to include at least 1024 MB plus the largest expected JVM heap size value **-Xmx**.

See *Limiting Storage use above the bar in z/Architecture* for information about setting the **MEMLIMIT** parameter: http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/FLASH10165.

Setting LE runtime options

LE runtime options can affect both performance and storage usage. The optimum settings will vary according to the host product and the Java application itself, but it is important to have good general settings.

The LE runtime options are documented in *Language Environment Programming Reference (SA22-7562)* at http://publibz.boulder.ibm.com/epubs/pdf/ceea3180.pdf.

Java and other products that are written in C or C++ might have LE runtime options embedded in the main programs by using `#pragma runopts`. These options are chosen to provide suitable default values that assist the performance in a typical environment. Any runtime overrides that you set might alter these values in a way that degrades the performance of Java or the host product. The host product's documentation might provide details of the product's default settings. Changes to the product's `#pragma runopts` might occur as a result of version or release changes. For details of how LE chooses the order of precedence of its runtime options, refer to the *Language Environment Programming Guide (SA22-7561)* at http://publibz.boulder.ibm.com/epubs/pdf/ceea2180.pdf.

Use the LE runtime option `RPTOPTS(ON)` as an override to write the options that are in effect, to stderr on termination. See the host product documentation and the *Language Environment Programming Guide (SA22-7561)* at http://publibz.boulder.ibm.com/epubs/pdf/ceea2180.pdf for details of how to supply LE runtime overrides. Before creating runtime overrides, run the application without overrides, to determine the existing options based on LE defaults and `#pragma` settings.

To tune the options, use the LE runtime option `RPTSTG(ON)` as an override, but be aware that performance could be reduced when you use this option. The output for `RPTSTG(ON)` also goes to stderr on termination. The *Language Environment Debugging Guide (GA22-7560)* at http://publibz.boulder.ibm.com/epubs/pdf/ceea1180.pdf explains `RPTSTG(ON)` output.

Setting LE 31-bit runtime options

There are a number of LE 31-bit options that are important for successful Java operation.

These options are as follows:

- **ANYHEAP**
- **HEAP**
- **HEAPPOOLS**
- **STACK**
- **STORAGE**
- **THREADSTACK**

You can change any, or all, of these options, however if you set the wrong values this might affect performance. The following values are a suggested starting point for these options:

```
ANYHEAP(2M,1M,ANY,FREE)
HEAP(80M,4M,ANY,KEEP)
HEAPPOOLS(ON,8,10,32,10,128,10,256,10,1024,10,2048,10,0,10,0, 10,0,10,0,10,0,10,0,10)
STACK(64K,16K,ANY,KEEP,128K,128K)
STORAGE(NONE,NONE,NONE,0K)
THREADSTACK(OFF,64K,16K,ANY,KEEP,128K,128K)
```

ANYHEAP and **HEAP** initial allocations (parameter 1) might be too large for transaction-based systems such as CICS®. Java applications that use many hundreds of threads might need to adjust the **STACK** initial and increment allocations (parameters 1, 2, 5 and 6) based on the **RPTSTG(ON)** output, which shows the maximum stack sizes that are used by a thread inside the application.

HEAPPOOLS(ON) should improve performance, but the LE-supplied default settings for the cell size and percentage pairs are not optimized for the best performance or storage usage.

For additional information, including how to set the LE runtime options, see:

- the *z/OS Language Environment Programming Reference (SA22-7562)* at http://publibz.boulder.ibm.com/epubs/pdf/ceea3180.pdf
- the *z/OS Language Environment Programming Guide (SA22-7561)* at http://publibz.boulder.ibm.com/epubs/pdf/ceea2180.pdf
- the host product documentation

Setting LE 64-bit runtime options

There are 64-bit versions of some of the runtime options.

These 64-bit options are as follows:

- **HEAP64**
- **HEAPPOOLS64**
- **STACK64**
- **THREADSTACK64**

A suggested start point for HEAP64 as an override is `HEAP64(512M,4M,KEEP,16M,4M,KEEP,0K,0K,FREE)`.

The following LE defaults should be appropriate:

```
STACK64(1M,1M,128M) THREADSTACK64(OFF,1M,1M,128M)
HEAPPOOLS64(OFF,8,4000,32,2000,128,700,256,350.1024,100,2048,50,
                    3072,50,4096,50,8192,25,16384,10,32768,5,65536,5)
```

Before you set an override for HEAPPOOLS64, use **RPTOPTS(ON)** or **RPTSTG(ON)** and check the result of `#pragma runopts`. Check this because the host product might have already set cell sizes and numbers that are known to produce good performance.

Also, these settings are dependant on a suitable **MEMLIMIT** setting. Based on these suggested LE 64-bit runtime options, the JVM requirement is a minimum of 512 MB as set for **HEAP64** (which should include **HEAPPOOLS64**), plus an initial value for **STACK64** of 1 MB times the expected maximum number of concurrent threads, plus the largest expected JVM heap **-Xmx** value.

Marking failures

The Java launcher can mark the z/OS Task Control Block (TCB) with an abend code when the launcher fails to load the VM or is terminated by an uncaught exception. To start TCB marking, set the environment variable **IBM_JAVA_ABEND_ON_FAILURE=Y**.

By default, the Java launcher will not mark the TCB.

Setting the path

If you alter the **PATH** environment variable, you will override any existing Java launchers in your path.

About this task

The **PATH** environment variable enables z/OS to find programs and utilities, such as **javac**, **java**, and **javadoc** tool, from any current directory. To display the current value of your **PATH**, type the following command at a command prompt:

```
echo $PATH
```

To add the Java launchers to your path:

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

   ```
   export PATH=/usr/lpp/java/J7.0[_64]/bin:/usr/lpp/java/J7.0[_64]/jre/bin:$PATH
   ```

2. Log on again or run the updated shell script to activate the new **PATH** environment variable.

Results

After setting the path, you can run a tool by typing its name at a command prompt from any directory. For example, to compile the file `Myfile.Java`, at a command prompt, type:

```
javac Myfile.Java
```

Setting the class path

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

About this task

You should set the class path explicitly only if:

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

To display the current value of your **CLASSPATH** environment variable, type the following command at a shell prompt:

```
echo $CLASSPATH
```

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

Updating your SDK or JRE for daylight saving time changes

You can apply recent changes to daylight saving time using the IBM Time Zone Update Utility for Java (JTZU).

About this task

Many countries around the world use a daylight saving time (DST) convention. Typically, clocks move forward by one hour during the summer months to create more daylight hours during the afternoon and less during the morning. This practice has many implications, including the need to adjust system clocks in computer systems. Occasionally, countries change their DST start and end dates. These changes can affect the date and time functions in applications, because the original start and end dates are programmed into the operating system and in Java software. To avoid this problem you must update operating systems and Java installations with the new DST information.

The Olson time zone database is an external resource that compiles information about the time zones around the world. This database establishes standard names for time zones, such as "America/New_York", and provides regular updates to time zone information that can be used as reference data. To ensure that Java JREs and SDKs contain up to date DST information, IBM incorporates the latest Olson update into each Java service refresh. To find out which Olson time zone update is included for a particular service refresh, see https://www.ibm.com/developerworks/java/jdk/dst/olson_table.html.

If a DST change has been introduced since the last service refresh, you can use JTZU to directly update your Java installation. You can also use this tool to update your installation if you are unable to move straight to the latest service refresh. JTZU is available from IBM developerWorks using the following link: https://www.ibm.com/developerworks/java/jdk/dst/jtzu.html.

Results

After updating your Java installation with any recent DST changes, your application can handle time and date calculations correctly.

Running the JVM under a different code page

Java for z/OS is shipped in only one version, for `EN_US`. To run Java under a different locale, convert all the text files in your Java installation from the default (IBM-1047) to your code page.

The **jdkconv** utility converts text files to a local encoding. The utility might be helpful if you get error messages about an invalid format for text files. To check if your code page setting might be the cause of the problem, find which locale you are using by checking the environment variables **LANG** or **LC_ALL**. If the locale value is not `C` or `EN_US` then you might see the invalid format message.

To convert your Java installation to a different code page, use the **jdkconv** utility. The utility requires a tool called **cpmod**, which is also provided in your Java installation.

1. Make a copy of your Java installation directory. The reason is that the script overwrites files in the Java installation directory. If you want to undo the changes, you must either reinstall Java, or restore your copy of the directory.
2. The **jdkconv** utility itself is shipped in code page IBM-1047. Before you run the utility, convert it to your code page as follows:

```
cp -p jdkconv jdkconv.backup
iconv -f IBM-1047 -t CODEPAGE jdkconv.backup >jdkconv
chmod 755 jdkconv
```

In this command sequence, replace ***CODEPAGE*** with your code page.

3. Ensure that the directory containing both **jdkconv** and **cpmod** is in your **PATH** setting.
4. Run **jdkconv** as follows:

   ```
   jdkconv CODEPAGE JAVATREE
   ```

   In this command, ***CODEPAGE*** is your code page, and ***JAVATREE*** is your Java installation directory.

After running the **jdkconv** utility, test the changes by running a Java application that is sensitive to the system locale.

Using non-default system fonts

If your operating system is z/OS V1.13 or earlier, you must install fonts separately, and optionally edit the Java fonts configuration properties file. For z/OS V2.1 and later, fonts are provided with the operating system by default, and no configuration is required.

About this task

Earlier releases of the z/OS operating system did not include font packages. From V2.1, fonts from the following packages are included by default, in the `/usr/lpp/fonts/worldtype` directory:

- AFP Font Collection for S/390™ (5648-B33)
- IBM Infoprint Fonts for z/OS V1.1 (5648-E76)
- World Type fonts that were not previously available in the z/OS operating system but form part of the InfoPrint Font Collection V3.1

Procedure

Complete these steps if you have z/OS V 1.13 or earlier. Later versions do not require configuration.

1. Purchase the font packages that you require. For example, you could purchase Infoprint WorldType Fonts for AFP Clients as part of program 5648-E77.
2. Install the font packages. Further configuration depends on where you install the packages:
   - If you install into the following directory, you do not have to do any further configuration, but the fonts are removed when you apply a service refresh, and must be replaced afterwards:
     - For IBM SDK Java Technology Edition, V7 service refresh 5 and earlier: *lib_dir*`/fonts`
     - For IBM SDK Java Technology Edition, V7 service refresh 6 and later: `/usr/lpp/fonts/worldtype`
   - If you install to a location outside *install_dir*, the fonts are unaffected when you apply a service refresh. However, you must edit the Java font configuration properties file, which is affected by service refreshes, so changes that you make must be reapplied.

   Note: These instructions assume that the font files are called `mtsansdj.ttf` and `tnrwt_j.ttf`. Depending on the font package that you purchased, your font file names might be different, in which case you must rename the files. For an example, see PM05140: Missing fonts on z/OS.

3. If you installed in a directory other than *lib_dir*/fonts or /usr/lpp/fonts/worldtype, modify the *lib_dir*/fontconfig.properties.src file by changing the paths in the following section:

```
 # Font File Names
filename.-Monotype-TimesNewRomanWT-medium-r-normal--*-%d-75-75-*
-*-jisx0208.1983-0=font_path/tnrwt_j.ttf
filename.-Monotype-SansMonoWT-medium-r-normal--*-%d-75-75-*-*-ji
sx0208.1983-0=font_path/mtsansdj.ttf
```

   Where *font_path* depends on your level of IBM SDK Java Technology Edition, V7:

   - For IBM SDK Java Technology Edition, V7 service refresh 5 and earlier: $JRE_LIB_FONTS
   - For IBM SDK Java Technology Edition, V7 service refresh 6 and later: /usr/lpp/fonts/worldtype

   Change the paths to match your font install location, then save a copy of the properties file so you can reapply the changes after you upgrade to a new service refresh.

Related information:

Information for Acquiring and Installing Fonts for z/OS Java

Chapter 5. Developing Java applications

The SDK contains many tools and libraries required for Java software development.

See "IBM Software Developers Kit (SDK)" on page 1 for details of the tools available.

Using XML

The IBM SDK contains the XML4J and XL XP-J parsers, the XL TXE-J 1.0 XSLT compiler, and the XSLT4J XSLT interpreter. These tools allow you to parse, validate, transform, and serialize XML documents independently from any given XML processing implementation.

Use factory finders to locate implementations of the abstract factory classes, as described in "Selecting an XML processor" on page 114. By using factory finders, you can select a different XML library without changing your Java code.

Available XML libraries

The IBM SDK for Java contains the following XML libraries:

XML4J 4.5

XML4J is a validating parser providing support for the following standards:

- XML 1.0 (4th edition)
- Namespaces in XML 1.0 (2nd edition)
- XML 1.1 (2nd edition)
- Namespaces in XML 1.1 (2nd edition)
- W3C XML Schema 1.0 (2nd Edition)
- XInclude 1.0 (2nd Edition)
- OASIS XML Catalogs 1.0
- SAX 2.0.2
- DOM Level 3 Core, Load and Save
- DOM Level 2 Core, Events, Traversal and Range
- JAXP 1.4

XML4J 4.5 is based on Apache Xerces-J 2.9.0. See http://xerces.apache.org/xerces2-j/ for more information.

XL XP-J 1.1

XL XP-J 1.1 is a high-performance non-validating parser that provides support for StAX 1.0 (JSR 173). StAX is a bidirectional API for pull-parsing and streaming serialization of XML 1.0 and XML 1.1 documents. See the "XL XP-J reference information" on page 118 section for more details about what is supported by XL XP-J 1.1.

XL TXE-J 1.0

For Version 5.0, the IBM SDK for Java included the XSLT4J compiler and interpreter. The XSLT4J interpreter was used by default.

For Version 6 and later, the IBM SDK for Java includes XL TXE-J. XL TXE-J includes the XSLT4J 2.7.8 interpreter and a new XSLT compiler. The new compiler is used by default. The XSLT4J compiler is no longer included with the IBM SDK for Java. See "Migrating to the XL-TXE-J" on page 115 for information about migrating to XL TXE-J.

XL TXE-J provides support for the following standards:

- XSLT 1.0
- XPath 1.0
- JAXP 1.4

Selecting an XML processor

XML processor selection is performed using service providers. When using a factory finder, Java looks in the following places, in this order, to see which service provider to use:

1. The system property with the same name as the service provider.
2. The service provider specified in a properties file.
   - **For XMLEventFactory, XMLInputFactory, and XMLOutputFactory only.** The value of the service provider in the file `/usr/lpp/java/J7.0[_64]/jre/lib/stax.properties`.
   - **For other factories.** The value of the service provider in the file `/usr/lpp/java/J7.0[_64]/jre/lib/jaxp.properties`.
3. The contents of the `META-INF/services/<service.provider>` file.
4. The default service provider.

The following service providers control the XML processing libraries used by Java:

`javax.xml.parsers.SAXParserFactory`
Selects the SAX parser. By default, org.apache.xerces.jaxp.SAXParserFactoryImpl from the XML4J library is used.

`javax.xml.parsers.DocumentBuilderFactory`
Selects the document builder. By default, org.apache.xerces.jaxp.DocumentBuilderFactoryImpl from the XML4J library is used.

`javax.xml.datatype.DatatypeFactory`
Selects the datatype factory. By default, org.apache.xerces.jaxp.datatype.DatatypeFactoryImpl from the XML4J library is used.

`javax.xml.stream.XMLEventFactory`
Selects the StAX event factory. By default, com.ibm.xml.xlxp.api.stax.XMLEventFactoryImpl from the XL XP-J library is used.

`javax.xml.stream.XMLInputFactory`
Selects the StAX parser. By default, com.ibm.xml.xlxp.api.stax.XMLInputFactoryImpl from the XL XP-J library is used.

javax.xml.stream.XMLOutputFactory
Selects the StAX serializer. By default, com.ibm.xml.xlxp.api.stax.XMLOutputFactoryImpl from the XL XP-J library is used.

javax.xml.transform.TransformerFactory
Selects the XSLT processor. Possible values are:

com.ibm.xtq.xslt.jaxp.compiler.TransformerFactoryImpl
Use the XL TXE-J compiler. This value is the default.

org.apache.xalan.processor.TransformerFactoryImpl
Use the XSLT4J interpreter.

javax.xml.validation.SchemaFactory:http://www.w3.org/2001/XMLSchema
Selects the schema factory for the W3C XML Schema language. By default, org.apache.xerces.jaxp.validation.XMLSchemaFactory from the XML4J library is used.

javax.xml.xpath.XPathFactory
Selects the XPath processor. By default, org.apache.xpath.jaxp.XPathFactoryImpl from the XSLT4J library is used.

Migrating to the XL-TXE-J

From Version 6, the XL TXE-J compiler replaces the XSLT4J interpreter as the default XSLT processor. If you are migrating applications from older versions of Java, follow these steps to prepare your application for the new library.

About this task

The XL TXE-J compiler is faster than the XSLT4J interpreter when you are applying the same transformation more than once. If you perform each individual transformation only once, the XL TXE-J compiler is slower than the XSLT4J interpreter because compilation and optimization reduce performance.

To continue using the XSLT4J interpreter as your XSLT processor, set the **javax.xml.transform.TransformerFactory** service provider to org.apache.xalan.processor.TransformerFactoryImpl.

To migrate to the XL-TXE-J compiler, follow the instructions in this task.

Procedure

1. Use com.ibm.xtq.xslt.jaxp.compiler.TransformerFactoryImpl when setting the **javax.xml.transform.TransformerFactory** service provider.
2. Regenerate class files generated by the XSLT4J compiler. XL TXE-J cannot execute class files generated by the XSLT4J compiler.
3. Some methods generated by the compiler might exceed the JVM method size limit, in which case the compiler attempts to split these methods into smaller methods.
   - If the compiler splits the method successfully, you receive the following warning:

     ```
     Some generated functions exceeded the JVM method size limit and were
     automatically split into smaller functions. You might get better
     performance by manually splitting very large templates into smaller
     templates, by using the 'splitlimit' option to the Process or Compile
     command, or by setting the 'http://www.ibm.com/xmlns/prod/xltxe-j/
     ```

`split-limit' transformer factory attribute.`You can use the compiled classes, but you might get better performance by controlling the split limit manually.

- If the compiler does not split the method successfully, you receive one of the following exceptions:

  `com.ibm.xtq.bcel.generic.ClassGenException: Branch target offset too large for short` or

  `bytecode array size > 65535 at offset=#####`Try setting the split limit manually, or decreasing the split limit.

To set the split limit, use the **-SPLITLIMIT** option when using the **Process** or **Compile** commands, or the **http://www.ibm.com/xmlns/prod/xltxe-j/split-limit** transformer factory attribute when using the transformer factory. The split limit can be between 100 and 2000. When setting the split limit manually, use the highest split limit possible for best performance.

4. XL TXE-J might need more memory than the XSLT4J compiler. If you are running out of memory or performance seems slow, increase the size of the heap using the **-Xmx** option.
5. Migrate your application to use the new attribute keys. The old transformer factory attribute keys are deprecated. The old names are accepted with a warning.

Table 3. Changes to attribute keys from the XSL4J compiler to the XL TXE-J compiler

XSL4J compiler attribute	XL TXE-J compiler attribute
translet-name	http://www.ibm.com/xmlns/prod/xltxe-j/translet-name
destination-directory	http://www.ibm.com/xmlns/prod/xltxe-j/destination-directory
package-name	http://www.ibm.com/xmlns/prod/xltxe-j/package-name
jar-name	http://www.ibm.com/xmlns/prod/xltxe-j/jar-name
generate-translet	http://www.ibm.com/xmlns/prod/xltxe-j/generate-translet
auto-translet	http://www.ibm.com/xmlns/prod/xltxe-j/auto-translet
use-classpath	http://www.ibm.com/xmlns/prod/xltxe-j/use-classpath
debug	http://www.ibm.com/xmlns/prod/xltxe-j/debug
indent-number	http://www.ibm.com/xmlns/prod/xltxe-j/indent-number
enable-inlining	*Obsolete in new compiler*

6. Optional: For best performance, ensure that you are not recompiling XSLT transformations that can be reused. Use one of the following methods to reuse compiled transformations:
   - If your stylesheet does not change at run time, compile the stylesheet as part of your build process and put the compiled classes on your classpath. Use the org.apache.xalan.xsltc.cmdline.Compile command to compile the stylesheet and set the **http://www.ibm.com/xmlns/prod/xltxe-j/use-classpath** transformer factory attribute to `true` to load the classes from the classpath.
   - If your application will use the same stylesheet during multiple runs, set the **http://www.ibm.com/xmlns/prod/xltxe-j/auto-translet** transformer factory attribute to `true` to automatically save the compiled stylesheet to disk for reuse. The compiler will use a compiled stylesheet if it is available, and compile the stylesheet if it is not available or is out-of-date. Use the **http://www.ibm.com/xmlns/prod/xltxe-j/destination-directory** transformer

factory attribute to set the directory used to store compiled stylesheets. By default, compiled stylesheets are stored in the same directory as the stylesheet.

- If your application is a long-running application that reuses the same stylesheet, use the transformer factory to compile the stylesheet and create a Templates object. You can use the Templates object to create Transformer objects without recompiling the stylesheet. The Transformer objects can also be reused but are not thread-safe.
- If your application uses each stylesheet just once or a very small number of times, or you are unable to make any of the other changes listed in this step, you might want to continue to use the XSLT4J interpreter by setting the javax.xml.transform.TransformerFactory service provider to org.apache.xalan.processor.TransformerFactoryImpl.

Securing 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. If you specify limits, you must override the default XML parser configuration with a custom configuration.

About this task

To protect your application from malformed data, you can enforce specific limits during JAXP processing. These limits can be set in your `jaxp.properties` file, or by specifying various system properties on the command line. However, for these limits to take effect you must also override the default XML parser configuration with a custom configuration that allows these secure processing limits.

Procedure

1. Select the limits that you want to set for your application.
   a. To limit the number of entity expansions in an XML document, see "-Djdk.xml.entityExpansionLimit" on page 424.
   b. To limit the maximum size of a general entity, see "-Djdk.xml.maxGeneralEntitySizeLimit" on page 424.
   c. To limit the maximum size of a parameter entity, see "-Djdk.xml.maxParameterEntitySizeLimit" on page 425
   d. To limit the total size of all entities that include general and parameter entities, see "-Djdk.xml.totalEntitySizeLimit" on page 426
   e. To define the maximum number of content model nodes that can be created in a grammar, see "-Djdk.xml.maxOccur" on page 425
2. To override the default XML parser configuration, set the custom configuration by specifying the following system property on the command line: **`-Dorg.apache.xerces.xni.parser.XMLParserConfiguration=`***config_file*, where *config_file* is **`org.apache.xerces.parsers.SecureProcessingConfiguration`**. For more information about the full override mechanism, see http://xerces.apache.org/xerces2-j/faq-xni.html#faq-2.

XML reference information

The XL XP-J and XL TXE-J XML libraries are new for Version 6 of the SDK. This reference information describes the features supported by these libraries.

XL XP-J reference information

XL XP-J 1.1 is a high-performance non-validating parser that provides support for StAX 1.0 (JSR 173). StAX is a bidirectional API for pull-parsing and streaming serialization of XML 1.0 and XML 1.1 documents.

Unsupported features

The following optional StAX features are not supported by XL XP-J:

- DTD validation when using an XMLStreamReader or XMLEventReader. The XL XP-J parser is non-validating.
- When using an XMLStreamReader to read from a character stream (java.io.Reader), the Location.getCharaterOffset() method always returns -1. The Location.getCharaterOffset() returns the byte offset of a Location when using an XMLStreamReader to read from a byte stream (java.io.InputStream).

XMLInputFactory reference

The javax.xml.stream.XMLInputFactory implementation supports the following properties, as described in the XMLInputFactory Javadoc information: http://download.oracle.com/javase/7/docs/api/javax/xml/stream/XMLInputFactory.html.

Property name	Supported?
javax.xml.stream.isValidating	No. The XL XP-J scanner does not support validation.
javax.xml.stream.isNamespaceAware	Yes, supports true and false. For XMLStreamReaders created from DOMSources, namespace processing depends on the methods that were used to create the DOM tree, and this value has no effect.
javax.xml.stream.isCoalescing	Yes
javax.xml.stream.isReplacingEntityReferences	Yes. For XMLStreamReaders created from DOMSources, if entities have already been replaced in the DOM tree, setting this parameter has no effect.
javax.xml.stream.isSupportingExternalEntities	Yes
javax.xml.stream.supportDTD	True is always supported. Setting the value to false works only if the **com.ibm.xml.xlxp.support.dtd.compat.mode** system property is also set to false. When both properties are set to false, parsers created by the factory throw an XMLStreamException when they encounter an entity reference that requires expansion. This setting is useful for protecting against Denial of Service (DoS) attacks involving entities declared in the DTD. Setting the value to false does not work before Service Refresh 2.
javax.xml.stream.reporter	Yes
javax.xml.stream.resolver	Yes

XL XP-J also supports the optional method createXMLStreamReader(javax.xml.transform.Source), which allows StAX readers to be created from DOM and SAX sources.

XL XP-J also supports the **javax.xml.stream.isSupportingLocationCoordinates** property. If you set this property to true, XMLStreamReaders created by the factory return accurate line, column, and character information using Location objects. If you set this property to false, line, column, and character information is not available. By default, this property is set to false for performance reasons.

XMLStreamReader reference

The javax.xml.stream.XMLStreamReader implementation supports the following properties, as described in the XMLStreamReader Javadoc: http://download.oracle.com/javase/7/docs/api/javax/xml/stream/XMLStreamReader.html.

Property name	Supported?
javax.xml.stream.entities	Yes
javax.xml.stream.notations	Yes

XL XP-J also supports the **javax.xml.stream.isInterning** property. This property returns a boolean value indicating whether or not XML names and namespace URIs returned by the API calls have been interned by the parser. This property is read-only.

XMLOutputFactory reference

The javax.xml.stream.XMLOutputFactory implementation supports the following properties, as described in the XMLOutputFactory Javadoc: http://download.oracle.com/javase/7/docs/api/javax/xml/stream/XMLOutputFactory.html.

Property name	Supported?
javax.xml.stream.isRepairingNamespaces	Yes

XL XP-J also supports the **javax.xml.stream.XMLOutputFactory.recycleWritersOnEndDocument** property. If you set this property to true, XMLStreamWriters created by this factory are recycled when writeEndDocument() is called. After recycling, some XMLStreamWriter methods, such as getNamespaceContext(), must not be called. By default, XMLStreamWriters are recycled when close() is called. You must call the XMLStreamWriter.close() method when you have finished with an XMLStreamWriter, even if this property is set to true.

XMLStreamWriter reference

The javax.xml.stream.XMLStreamWriter implementation supports the following properties, as described in the XMLStreamWriter Javadoc: http://download.oracle.com/javase/7/docs/api/javax/xml/stream/XMLStreamWriter.html.

Property name	Supported?
javax.xml.stream.isRepairingNamespaces	Yes

Properties on XMLStreamWriter objects are read-only.

XL XP-J also supports the **javax.xml.stream.XMLStreamWriter.isSetPrefixBeforeStartElement** property. This property returns a Boolean indicating whether calls to setPrefix() and setDefaultNamespace() should occur before calls to writeStartElement() or writeEmptyElement() to put a namespace prefix in scope for that element. XL XP-J always returns false; calls to setPrefix() and setDefaultNamespace() should occur after writeStartElement() or writeEmptyElement().

XL TXE-J reference information

XL TXE-J is an XSLT library containing the XSLT4J 2.7.8 interpreter and a XSLT compiler.

Feature comparison table

Table 4. Comparison of the features in the XSLT4J interpreter, the XSLT4J compiler, and the XL TXE-J compiler.

Feature	XSLT4J interpreter (included)	XSLT4J compiler (not included)	XL TXE-J compiler (included)
http://javax.xml.transform.stream.StreamSource/feature feature	Yes	Yes	Yes
http://javax.xml.transform.stream.StreamResult/feature feature	Yes	Yes	Yes
http://javax.xml.transform.dom.DOMSource/feature feature	Yes	Yes	Yes
http://javax.xml.transform.dom.DOMResult/feature feature	Yes	Yes	Yes
http://javax.xml.transform.sax.SAXSource/feature feature	Yes	Yes	Yes
http://javax.xml.transform.sax.SAXResult/feature feature	Yes	Yes	Yes
http://javax.xml.transform.stax.StAXSource/feature feature	Yes	No	Yes
http://javax.xml.transform.stax.StAXResult/feature feature	Yes	No	Yes
http://javax.xml.transform.sax.SAXTransformerFactory/feature feature	Yes	Yes	Yes
http://javax.xml.transform.sax.SAXTransformerFactory/feature/xmlfilter feature	Yes	Yes	Yes
http://javax.xml.XMLConstants/feature/secure-processing feature	Yes	Yes	Yes
http://xml.apache.org/xalan/features/incremental attribute	Yes	No	No
http://xml.apache.org/xalan/features/optimize attribute	Yes	No	No
http://xml.apache.org/xalan/properties/source-location attribute	Yes	No	No
translet-name attribute	N/A	Yes	Yes (with new name)
destination-directory attribute	N/A	Yes	Yes (with new name)

Table 4. Comparison of the features in the XSLT4J interpreter, the XSLT4J compiler, and the XL TXE-J compiler. (continued)

Feature	XSLT4J interpreter (included)	XSLT4J compiler (not included)	XL TXE-J compiler (included)
package-name attribute	N/A	Yes	Yes (with new name)
jar-name attribute	N/A	Yes	Yes (with new name)
generate-translet attribute	N/A	Yes	Yes (with new name)
auto-translet attribute	N/A	Yes	Yes (with new name)
use-classpath attribute	N/A	Yes	Yes (with new name)
enable-inlining attribute	No	Yes	No (obsolete in TL TXE-J)
indent-number attribute	No	Yes	Yes (with new name)
debug attribute	No	Yes	Yes (with new name)
Java extensions	Yes	Yes (abbreviated syntax only, xalan:component/ xalan:script constructs not supported)	Yes (abbreviated syntax only, xalan:component/ xalan:script constructs not supported)
JavaScript extensions	Yes	No	No
Extension elements	Yes	No	No
EXSLT extension functions	Yes	Yes (excluding dynamic)	Yes (excluding dynamic)
redirect extension	Yes	Yes (excluding redirect:open and redirect:close)	Yes
output extension	No	Yes	Yes
nodeset extension	Yes	Yes	Yes
NodeInfo extension functions	Yes	No	No
SQL library extension	Yes	No	No
pipeDocument extension	Yes	No	No
evaluate extension	Yes	No	No
tokenize extension	Yes	No	No
XML 1.1	Yes	Yes	Yes

Notes

1. With the Process command, use **-FLAVOR sr2sw** to transform using StAX stream processing, and **-FLAVOR er2ew** for StAX event processing.
2. The new compiler does not look for the org.apache.xalan.xsltc.dom.XSLTCDTMManager service provider. Instead, if StreamSource is used, the compiler switches to a high-performance XML parser.

3. Inlining is obsolete in XL TXE-J.
   - The **-XN** option to the **Process** command is silently ignored.
   - The **-n** option to the **Compile** command is silently ignored.
   - The **enable-inlining** transformer factory attribute is silently ignored.
4. The org.apache.xalan.xsltc.trax.SmartTransformerFactoryImpl class is no longer supported.

Using an older version of Xerces or Xalan

If you are using an older version of Xerces (before 2.0) or Xalan (before 2.3) in the endorsed override, you might get a NullPointerException when you start your application. This exception occurs because these older versions do not handle the `jaxp.properties` file correctly.

About this task

To avoid this situation, use one of the following workarounds:

- Upgrade to a newer version of the application that implements the latest Java API for XML Programming (JAXP) specification (https://jaxp.dev.java.net/).
- Remove the `jaxp.properties` file from `/usr/lpp/java/J7.0[_64]/jre/lib`.
- Uncomment the entries in the `jaxp.properties` file in `/usr/lpp/java/J7.0[_64]/jre/lib`.
- Set the system property for **javax.xml.parsers.SAXParserFactory**, **javax.xml.parsers.DocumentBuilderFactory**, or **javax.xml.transform.TransformerFactory** using the **-D** command-line option.
- Set the system property for **javax.xml.parsers.SAXParserFactory**, **javax.xml.parsers.DocumentBuilderFactory**, or **javax.xml.transform.TransformerFactory** in your application. For an example, see the JAXP 1.4 specification.
- Explicitly set the SAX parser, Document builder, or Transformer factory using the **IBM_JAVA_OPTIONS** environment variable.

```
export IBM_JAVA_OPTIONS=-Djavax.xml.parsers.SAXParserFactory=
  org.apache.xerces.jaxp.SAXParserFactoryImpl
```

or

```
export IBM_JAVA_OPTIONS=-Djavax.xml.parsers.DocumentBuilderFactory=
  org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
```

or

```
export IBM_JAVA_OPTIONS=-Djavax.xml.transform.TransformerFactory=
  org.apache.xalan.processor.TransformerFactoryImpl
```

Debugging Java applications

To debug Java programs, you can use the Java Debugger (JDB) application or other debuggers that communicate by using the Java Platform Debugger Architecture (JPDA) that is provided by the SDK for the operating system.

More information about problem diagnosis using Java can be found in the Diagnostic GuideChapter 9, “Troubleshooting and support,” on page 173.

Java Debugger (JDB)

The Java Debugger (JDB) is included in the SDK. The debugger is started with the **jdb** command; it attaches to the JVM using JPDA.

To debug a Java application:

1. Start the JVM with the following options:

   ```
   java -agentlib:jdwp=transport=dt_socket,server=y,address=<port> <class>
   ```

   The JVM starts up, but suspends execution before it starts the Java application.
2. In a separate session, you can attach the debugger to the JVM:

   ```
   jdb -attach <port>
   ```

   The debugger will attach to the JVM, and you can now issue a range of commands to examine and control the Java application; for example, type `run` to allow the Java application to start.

For more information about JDB options, type:

```
jdb -help
```

For more information about JDB commands:

1. Type `jdb`
2. At the **jdb** prompt, type `help`

You can also use JDB to debug Java applications running on remote workstations. JPDA uses a TCP/IP socket to connect to the remote JVM.

1. Start the JVM with the following options:

   ```
   java -agentlib:jdwp=transport=dt_socket,server=y,address=<port> <class>
   ```

   The JVM starts up, but suspends execution before it starts the Java application.
2. Attach the debugger to the remote JVM:

   ```
   jdb -attach <host>:<port>
   ```

The Java Virtual Machine Debugging Interface (JVMDI) is not supported in this release. It has been replaced by the Java Virtual Machine Tool Interface (JVMTI).

For more information about JDB and JPDA and their usage, see these Web sites:

- http://www.oracle.com/technetwork/java/javase/tech/jpda-141715.html
- http://download.oracle.com/javase/7/docs/technotes/guides/jpda/
- http://download.oracle.com/javase/7/docs/technotes/guides/jpda/jdb.html

Determining whether your application is running on a 31-bit or 64-bit JVM

Some Java applications must be able to determine whether they are running on a 31-bit JVM or on a 64-bit JVM. For example, if your application has a native code library, the library must be compiled separately in 31- and 64-bit forms for platforms that support both 31- and 64-bit modes of operation. In this case, your application must load the correct library at run environmenttime, because it is not possible to mix 31- and 64-bit code.

About this task

The system property **com.ibm.vm.bitmode** allows applications to determine the mode in which your JVM is running. It returns the following values:

- 32 - the JVM is running in 31-bit mode
- 64 - the JVM is running in 64-bit mode

You can inspect the **com.ibm.vm.bitmode** property from inside your application code using the call:

```
System.getProperty("com.ibm.vm.bitmode");
```

How the JVM processes signals

When a signal is raised that is of interest to the JVM, a signal handler is called. This signal handler determines whether it has been called for a Java or non-Java thread.

If the signal is for a Java thread, the JVM takes control of the signal handling. If an application handler for this signal is installed and you did not specify the **-Xnosigchain** command-line option, the application handler for this signal is called after the JVM has finished processing.

If the signal is for a non-Java thread, and the application that installed the JVM had previously installed its own handler for the signal, control is given to that handler. Otherwise, if the signal is requested by the JVM or Java application, the signal is ignored or the default action is taken.

For exception and error signals, the JVM either:

- Handles the condition and recovers, or
- Enters a controlled shut down sequence where it:
  1. Produces dumps, to describe the JVM state at the point of failure
  2. Calls your application's signal handler for that signal
  3. Calls any application-installed unexpected shut down hook
  4. Performs the necessary JVM cleanup

For information about writing a launcher that specifies the previous hooks, see: http://www.ibm.com/developerworks/java/library/i-signalhandling/. This item was written for Java V1.3.1, but still applies to later versions.

For interrupt signals, the JVM also enters a controlled shut down sequence, but this time it is treated as a normal termination that:

1. Calls your application's signal handler for that signal
2. Calls all application shut down hooks
3. Calls any application-installed exit hook
4. Performs the necessary JVM cleanup

The shut down is identical to the shut down initiated by a call to the Java method `System.exit()`.

Other signals that are used by the JVM are for internal control purposes and do not cause it to stop. The only control signal of interest is SIGQUIT, which causes a Javadump to be generated.

Signals used by the JVM

The types of signals are Exceptions, Errors, Interrupts, and Controls.

Table 5 on page 125 shows the signals that are used by the JVM. The signals are grouped in the table by type or use, as follows:

Exceptions
The operating system synchronously raises an appropriate exception signal whenever an unrecoverable condition occurs.

Errors The JVM raises a SIGABRT if it detects a condition from which it cannot recover.

Interrupts
Interrupt signals are raised asynchronously, from outside a JVM process, to request shut down.

Controls
Other signals that are used by the JVM for control purposes.

Table 5. Signals used by the JVM

Signal Name	Signal type	Description	Disabled by -Xrs	Disabled by -Xrs:sync
SIGBUS (7)	Exception	Incorrect access to memory (data misalignment)	Yes	Yes
SIGSEGV (11)	Exception	Incorrect access to memory (write to inaccessible memory)	Yes	Yes
SIGILL (4)	Exception	Illegal instruction (attempt to call an unknown machine instruction)	Yes	Yes
SIGFPE (8)	Exception	Floating point exception (divide by zero)	Yes	Yes
SIGABRT (6)	Error	Abnormal termination. The JVM raises this signal whenever it detects a JVM fault.	Yes	Yes
SIGINT (2)	Interrupt	Interactive attention (CTRL-C). JVM exits normally.	Yes	No
SIGTERM (15)	Interrupt	Termination request. JVM will exit normally.	Yes	No
SIGHUP (1)	Interrupt	Hang up. JVM exits normally.	Yes	No
SIGQUIT (3)	Control	By default, this triggers a Javadump.	Yes	No

Table 5. Signals used by the JVM (continued)

Signal Name	Signal type	Description	Disabled by -Xrs	Disabled by -Xrs:sync
SIGRECONFIG (58)	Control	Reserved to detect any change in the number of CPUs, processing capacity, or physical memory.	Yes	No
SIGTRAP (5)	Control	Used by the JIT.	Yes	Yes
SIGCHLD (17)	Control	Used by the SDK for internal control.	No	No
SIGUSR1	Control	Used by the SDK.	No	No

Note: A number supplied after the signal name is the standard numeric value for that signal.

Use the **-Xrs** (reduce signal usage) option to prevent the JVM from handling most signals. For more information, see Oracle's Java application launcher page.

Signals 1 (SIGHUP), 2 (SIGINT), 4 (SIGILL), 7 (SIGBUS), 8 (SIGFPE), 11 (SIGSEGV), and 15 (SIGTERM) on JVM threads cause the JVM to shut down; therefore, an application signal handler should not attempt to recover from these unless it no longer requires the JVM.

Linking a native code driver to the signal-chaining library

The Runtime Environment contains signal-chaining. Signal-chaining enables the JVM to interoperate more efficiently with native code that installs its own signal handlers.

About this task

Signal-chaining enables an application to link and load the shared library `libjsig.so` before the system libraries. The libjsig.so library ensures that calls such as signal(), sigset(), and sigaction() are intercepted so that their handlers do not replace the JVM's signal handlers. Instead, these calls save the new signal handlers, or "chain" them behind the handlers that are installed by the JVM. Later, when any of these signals are raised and found not to be targeted at the JVM, the preinstalled handlers are invoked.

If you install signal handlers that use sigaction() , some **sa_flags** are not observed when the JVM uses the signal. These are:

- SA_NOCLDSTOP - This is always unset.
- SA_NOCLDWAIT - This is always unset.
- SA_RESTART - This is always set.

The libjsig.so library also hides JVM signal handlers from the application. Therefore, calls such as signal(), sigset(), and sigaction() that are made after the

JVM has started no longer return a reference to the JVM's signal handler, but instead return any handler that was installed before JVM startup.

The environment variable **JAVA_HOME** should be set to the location of the SDK, for example,*install_dir*.

To use libjsig.a:

- Link it with the application that creates or embeds a JVM:

```
cc_r -q64 <other compile/link parameter> -Linstall_dir
-ljsig -Linstall_dir/jre/bin/j9vm -ljvm java_application.c
```

  Note: Use `xlc_r` or `xlC_r` in place of `cc_r` if that is how you usually call the compiler or linker.

Writing JNI applications

Valid Java Native Interface (JNI) version numbers that programs can specify on the JNI_CreateJavaVM() API call are: JNI_VERSION_1_2(0x00010002) and JNI_VERSION_1_4(0x00010004).

Restriction: Version 1.1 of the JNI is not supported.

This version number determines only the level of the JNI to use. The actual level of the JVM that is created is specified by the JSE libraries (use the `java -version` command to show the JVM level). The JNI level *does not* affect the language specification that is implemented by the JVM, the class library APIs, or any other area of JVM behavior. For more information, see http://download.oracle.com/javase/7/docs/technotes/guides/jni/.

If your application needs two JNI libraries, one built for 31- and the other for 64-bit, use the **com.ibm.vm.bitmode** system property to determine if you are running with a 31- or 64-bit JVM and choose the appropriate library.

For more information about writing 64-bit applications, see the IBM Redpaper *z/OS 64-bit C/C++ and Java Programming Environment* at http://www.redbooks.ibm.com/abstracts/redp9110.html.

ASCII and EBCDIC issues

On z/OS, the Java Virtual Machine is essentially an EBCDIC application. Enhanced ASCII methods are C or C++ code that has been compiled with ASCII compiler options. If you create JNI routines as enhanced ASCII C or C++ methods you will be operating in a bimodal environment; your application will be crossing over between ASCII and EBCDIC environments.

The inherent problem with bimodal programs is that, in the z/OS runtime environment, threads are designated as either EBCDIC or enhanced ASCII and are not intended to be switched between these modes in typical use. Enhanced ASCII is not designed to handle bimodal issues. You might get unexpected results or experience failures when the active mode does not match that of the compiled code. There are z/OS runtime calls that applications might use to switch the active mode between EBCDIC and enhanced ASCII (the __ae_thread_swapmode() and __ae_thread_setmode() functions are documented in Language Environment Vendor Interfaces, see the *SA22-7568-06 Red Book*: http://publibz.boulder.ibm.com/epubs/pdf/ceev1160.pdf). However, even if an application is carefully coded to switch modes correctly, other bimodal issues might exist.

Supported compilers

These compilers have been tested with the IBM SDK.

The c89 compiler packaged with z/OS v1.11, C/OS/390 C++ Optional Feature is supported for:

- 31-bit z/OS on S/390®
- 64-bit z/OS on S/390

Native formatting of Java types long, double, float

The latest C/C++ compilers and runtime environments can convert jlong, jdouble, and jfloat data types to strings by using printf()-type functions.

Previous versions of the SDK for z/OS 31-bit had a set of native conversion functions and macros for formatting large Java data types. These functions and macros were:

ll2str() function
Converts a jlong to an ASCII string representation of the 64-bit value.

flt2dbl() function
Converts a jfloat to a jdouble.

dbl2nat() macro
Converts a jdouble to an ESA/390 native double.

dbl_sqrt() macro
Calculates the square root of a jdouble and returns it as a jdouble.

dbl2str() function
Converts a jdouble to an ASCII string representation.

flt2str() function
Converts a jfloat to an ASCII string representation.

These functions and macros are no longer supported by Version 6 of the SDK for z/OS. To provide a migration path, the functions have been moved to the demos area of the SDK. The demo code for these functions has been updated to reflect the changes.

The functions ll2str(), dbl2str(), and flt2str() are provided in the following object files:

- *install_dir*/demo/jni/JNINativeTypes/c/convert.o (For 31-bit)
- *install_dir*/demo/jni/JNINativeTypes/c/convert64.o (For 64-bit)

The function flt2dbl() and the macros dbl2nat() and dbl_sqrt() are not defined. However, the following macros give their definitions:

```
#include <math.h>
#define flt2dbl(f)  ((double)f)
#define dbl2nat(a)  ((a))
#define dbl_sqrt(a) (sqrt(a))
```

A C/C++ application that returns a jfloat data type to a Java application must be compiled with the FLOAT (IEEE) C/C++ compiler option. Applications compiled without this option return incorrect data types. Further information about compiling C/C++ source code, which applies to this Java release, can be found in the article http://www.ibm.com/developerworks/java/library/j-jni/

Support for thread-level recovery of blocked connectors

Four new IBM-specific SDK classes have been added to the com.ibm.jvm package to support the thread-level recovery of Blocked connectors. The new classes are packaged in `core.jar`.

These classes allow you to unblock threads that have become blocked on networking or synchronization calls. If an application does not use these classes, it must end the whole process, rather than interrupting an individual blocked thread.

The classes are:

`public interface InterruptibleContext`
Defines two methods, isBlocked() and unblock(). The other three classes implement InterruptibleContext.

`public class InterruptibleLockContext`
A utility class for interrupting synchronization calls.

`public class InterruptibleIOContext`
A utility class for interrupting network calls.

`public class InterruptibleThread`
A utility class that extends java.lang.Thread, to allow wrapping of interruptible methods. It uses instances of InterruptibleLockContext and InterruptibleIOContext to perform the required isBlocked() and unblock() methods depending on whether a synchronization or networking operation is blocking the thread.

Both InterruptibleLockContext and InterruptibleIOContext work by referencing the current thread. Therefore if you do not use InterruptibleThread, you must provide your own class that extends java.lang.Thread, to use these new classes.

API documentation to support the package containing these classes is available here: API documentation

CORBA support

The Java Platform, Standard Edition (JSE) supports, at a minimum, the specifications that are defined in the compliance document from Oracle. In some cases, the IBM JSE ORB supports more recent versions of the specifications.

The minimum specifications supported are defined in the Official Specifications for CORBA support in Java SE 7: http://download.oracle.com/javase/7/docs/api/org/omg/CORBA/doc-files/compliance.html.

Support for GIOP 1.2

This SDK supports all versions of GIOP, as defined by chapters 13 and 15 of the CORBA 2.3.1 specification, OMG document *formal/99-10-07*.

http://www.omg.org/cgi-bin/doc?formal/99-10-07

Bidirectional GIOP is not supported.

Support for Portable Interceptors

This SDK supports Portable Interceptors, as defined by the OMG in the document *ptc/01–03–04*, which you can obtain from:

http://www.omg.org/cgi-bin/doc?ptc/01–03-04

Portable Interceptors are hooks into the ORB that ORB services can use to intercept the normal flow of execution of the ORB.

Support for Interoperable Naming Service

This SDK supports the Interoperable Naming Service, as defined by the OMG in the document *ptc/00-08-07*, which you can obtain from:

http://www.omg.org/cgi-bin/doc?ptc/00-08-07

The default port that is used by the Transient Name Server (the **tnameserv** command), when no **ORBInitialPort** parameter is given, has changed from *900* to *2809*, which is the port number that is registered with the IANA (Internet Assigned Number Authority) for a CORBA Naming Service. Programs that depend on this default might have to be updated to work with this version.

The initial context that is returned from the Transient Name Server is now an org.omg.CosNaming.NamingContextExt. Existing programs that narrow the reference to a context org.omg.CosNaming.NamingContext still work, and do not need to be recompiled.

The ORB supports the **-ORBInitRef** and **-ORBDefaultInitRef** parameters that are defined by the Interoperable Naming Service specification, and the ORB::string_to_object operation now supports the ObjectURL string formats (corbaloc: and corbaname:) that are defined by the Interoperable Naming Service specification.

The OMG specifies a method ORB::register_initial_reference to register a service with the Interoperable Naming Service. However, this method is not available in the Oracle Java Core API at this release. Programs that have to register a service in the current version must invoke this method on the IBM internal ORB implementation class. For example, to register a service “MyService”:

```
((com.ibm.CORBA.iiop.ORB)orb).register_initial_reference("MyService",
serviceRef);
```

Where `orb` is an instance of org.omg.CORBA.ORB, which is returned from ORB.init(), and `serviceRef` is a CORBA Object, which is connected to the ORB. This mechanism is an interim one, and is not compatible with future versions or portable to non-IBM ORBs.

System properties for tracing the ORB

A runtime debug feature provides improved serviceability. You might find it useful for problem diagnosis or it might be requested by IBM service personnel.

Tracing Properties

com.ibm.CORBA.Debug=*true*
Turns on ORB tracing.

com.ibm.CORBA.CommTrace=*true*
Adds GIOP messages (sent and received) to the trace.

com.ibm.CORBA.Debug.Output=<*file***>**
Specify the trace output file. By default, this is of the form `orbtrc.DDMMYYYY.HHmm.SS.txt`.

Example of ORB tracing

For example, to trace events and formatted GIOP messages from the command line, type:

```
java -Dcom.ibm.CORBA.Debug=true
     -Dcom.ibm.CORBA.CommTrace=true <myapp>
```

Limitations

Do not enable tracing for normal operation, because it might cause performance degradation. Even if you have switched off tracing, FFDC (First Failure Data Capture) is still working, so serious errors are reported. If a debug output file is generated, examine it to check on the problem. For example, the server might have stopped without performing an ORB.shutdown().

The content and format of the trace output might vary from version to version.

System properties for tuning the ORB

The ORB can be tuned to work well with your specific network. The properties required to tune the ORB are described here.

com.ibm.CORBA.FragmentSize=<*size in bytes***>**
Used to control GIOP 1.2 fragmentation. The default size is 1024 bytes.

To disable fragmentation, set the fragment size to 0 bytes:

```
java -Dcom.ibm.CORBA.FragmentSize=0 <myapp>
```

com.ibm.CORBA.RequestTimeout=<*time in seconds***>**
Sets the maximum time to wait for a CORBA Request. By default the ORB waits indefinitely. Do not set the timeout too low to avoid connections ending unnecessarily.

com.ibm.CORBA.LocateRequestTimeout=<*time in seconds***>**
Set the maximum time to wait for a CORBA LocateRequest. By default the ORB waits indefinitely.

com.ibm.CORBA.ListenerPort=<*port number***>**
Set the port for the ORB to read incoming requests on. If this property is set, the ORB starts listening as soon as it is initialized. Otherwise, it starts listening only when required.

Java security permissions for the ORB

When running with a Java SecurityManager, invocation of some methods in the CORBA API classes might cause permission checks to be made, which might result in a SecurityException. If your program uses any of these methods, ensure that it is granted the necessary permissions.

Table 6. Methods affected when running with Java SecurityManager

Class/Interface	Method	Required permission
org.omg.CORBA.ORB	init	java.net.SocketPermission resolve
org.omg.CORBA.ORB	connect	java.net.SocketPermission listen
org.omg.CORBA.ORB	resolve_initial_references	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_is_a	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_non_existent	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	OutputStream _request (String, boolean)	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_get_interface_def	java.net.SocketPermission connect
org.omg.CORBA. Request	invoke	java.net.SocketPermission connect
org.omg.CORBA. Request	send_deferred	java.net.SocketPermission connect
org.omg.CORBA. Request	send_oneway	java.net.SocketPermission connect
javax.rmi. PortableRemoteObject	narrow	java.net.SocketPermission connect

ORB implementation classes

A list of the ORB implementation classes.

The ORB implementation classes in this release are:
- org.omg.CORBA.ORBClass=com.ibm.CORBA.iiop.ORB
- org.omg.CORBA.ORBSingletonClass=com.ibm.rmi.corba.ORBSingleton
- javax.rmi.CORBA.UtilClass=com.ibm.CORBA.iiop.UtilDelegateImpl
- javax.rmi.CORBA.StubClass=com.ibm.rmi.javax.rmi.CORBA.StubDelegateImpl
- javax.rmi.CORBA.PortableRemoteObjectClass =com.ibm.rmi.javax.rmi.PortableRemoteObject

These are the default values, and you are advised not to set these properties or refer to the implementation classes directly. For portability, make references only to the CORBA API classes, and not to the implementation. These values might be changed in future releases.

RMI over IIOP

Java Remote Method Invocation (RMI) provides a simple mechanism for distributed Java programming. RMI over IIOP (RMI-IIOP) uses the Common Object Request Broker Architecture (CORBA) standard Internet Inter-ORB Protocol (IIOP) to extend the base Java RMI to perform communication. This allows direct interaction with any other CORBA Object Request Brokers (ORBs), whether they were implemented in Java or another programming language.

The following documentation is available:

- The *Java Language to IDL Mapping* document is a detailed technical specification of RMI-IIOP: http://www.omg.org/cgi-bin/doc?ptc/00-01-06.pdf.

RMI-IIOP Programmer's Guide

Discusses how to write Java Remote Method Invocation (RMI) programs that can access remote objects by using the Internet Inter-ORB Protocol (IIOP).

Background reading

Links to Web sites related to RMI and related technologies.

Here are some sites to help you with this technology:

- The Java RMI home page contains links to RMI documentation, examples, specification, and more: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html
- The RMI trail in the Java Tutorial: http://download.oracle.com/javase/tutorial/rmi/
- The RMI API Javadoc HTML contains the most up-to-date RMI API documentation: http://download.oracle.com/javase/7/docs/api/java/rmi/package-summary.html
- The Java IDL Web page will familiarize you with Oracle's CORBA/IIOP implementation: http://download.oracle.com/javase/7/docs/technotes/guides/idl/index.html
- The Java IDL Trail in the Java Tutorial: http://download.oracle.com/javase/7/docs/technotes/guides/idl/GShome.html

What are RMI, IIOP, and RMI-IIOP?

The basic concepts behind RMI-IIOP and other similar technologies.

RMI

With RMI, you can write distributed programs in the Java programming language. RMI is easy to use, you do not need to learn a separate interface definition language (IDL), and you get Java's inherent "write once, run anywhere" benefit. Clients, remote interfaces, and servers are written entirely in Java. RMI uses the Java Remote Method Protocol (JRMP) for remote Java object communication. For a quick introduction to writing RMI programs, see the RMI tutorial Web page: http://download.oracle.com/javase/tutorial/rmi/, which describes writing a simple "Hello World" RMI program.

RMI lacks interoperability with other languages, and, because it uses a non-standard communication protocol, cannot communicate with CORBA objects.

IIOP, CORBA, and Java IDL

IIOP is CORBA's communication protocol. It defines the way bits are sent over a wire between CORBA clients and servers. CORBA is a standard distributed object architecture developed by the Object Management Group (OMG). Interfaces to remote objects are described in a platform-neutral interface definition language (IDL). Mappings from IDL to specific programming languages are implemented, binding the language to CORBA/IIOP.

The Java Standard Edition CORBA/IIOP implementation is known as Java IDL. Along with the IDL to Java (idlj) compiler, Java IDL can be used to define, implement, and access CORBA objects from the Java programming language.

The Java IDL Web page: http://download.oracle.com/javase/1.5.0/docs/guide/idl/index.html, gives you a good, Java-centric view of CORBA/IIOP programming. To get a quick introduction to writing Java IDL programs, see the Getting Started: Hello World Web page: http://download.oracle.com/javase/1.5.0/docs/guide/idl/GShome.html.

RMI-IIOP

Previously, Java programmers had to choose between RMI and CORBA/IIOP (Java IDL) for distributed programming solutions. Now, by adhering to a few restrictions (see "Restrictions when running RMI programs over IIOP" on page 138), RMI server objects can use the IIOP protocol, and communicate with CORBA client objects written in any language. This solution is known as RMI-IIOP. RMI-IIOP combines RMI ease of use with CORBA cross-language interoperability.

Using RMI-IIOP

This section describes how to use the IBM RMI-IIOP implementation.

The rmic compiler:

Reference information about the **rmic** compiler.

Purpose

The **rmic** compiler generates IIOP stubs and ties, and emits IDL, in accordance with the Java Language to OMG IDL Language Mapping Specification: http://www.omg.org/cgi-bin/doc?formal/01-06-07.

Parameters

-iiop

Generates stub and tie classes. A stub class is a local proxy for a remote object. Clients use stub classes to send calls to a server. Each remote interface requires a stub class, which implements that remote interface. The remote object reference used by a client is a reference to a stub. Tie classes are used on the server side to process incoming calls, and dispatch the calls to the correct implementation class. Each implementation class requires a tie class.

Stub classes are also generated for abstract interfaces. An abstract interface is an interface that does not extend java.rmi.Remote, but has methods that throw either java.rmi.RemoteException or a superclass of java.rmi.RemoteException. Interfaces that do not extend java.rmi.Remote and have no methods are also abstract interfaces.

-poa

Changes the inheritance from org.omg.CORBA_2_3.portable.ObjectImpl to org.omg.PortableServer.Servant. This type of mapping is nonstandard and is not specified by the Java Language to OMG IDL Mapping Specification: http://www.omg.org/cgi-bin/doc?formal/01-06-07.

The PortableServer module for the Portable Object Adapter (POA) defines the native Servant type. In the Java programming language, the Servant type is mapped to the Java org.omg.PortableServer.Servant class. The class serves as the base class for all POA servant implementations. It provides a number of methods that can be called by the application programmer, as well as methods that are called by the POA itself and might be overridden by the user to control aspects of servant behavior.

Valid only when the **-iiop** option is present.

-idl

Generates OMG IDL for the classes specified and any classes referenced. This option is required only if you have a CORBA client written in another language that needs to talk to a Java RMI-IIOP server.

Tip: After the OMG IDL is generated using **rmic -idl**, use the generated IDL with an IDL-to-C++ or other language compiler, but not with the IDL-to-Java language compiler. "Round tripping" is not recommended and should not be necessary. The IDL generation facility is intended to be used with other languages. Java clients or servers can use the original RMI-IIOP types.

IDL provides a purely declarative means of specifying the API for an object. IDL is independent of the programming language used. The IDL is used as a specification for methods and data that can be written in and called from any language that provides CORBA bindings. Java and C++ are such languages. For a complete description, see the Java Language to OMG IDL Mapping Specification: http://www.omg.org/cgi-bin/doc?formal/01-06-07.

Restriction: The generated IDL can be compiled using only an IDL compiler that supports the CORBA 2.3 extensions to IDL.

-always

Forces regeneration even when existing stubs, ties, or IDL are newer than the input class. Valid only when **-iiop** or **-idl** options are present.

-noValueMethods

Ensures that methods and initializers are not included in valuetypes emitted during IDL Generation. Methods and initializers are optional for valuetypes and are otherwise omitted.

Only valid when used with **-idl** option.

-idlModule *<fromJavaPackage[.class]> <toIDLModule>*

Specifies IDLEntity package mapping. For example: `-idlModule sample.bar my::real::idlmod`.

Only valid when used with **-idl** option.

-idlFile *<fromJavaPackage[.class]> <toIDLModule>*

Specifies IDLEntity file mapping. For example: `-idlFile test.pkg.X TEST16.idl`.

Only valid when used with **-idl** option.

More Information

For more detailed information about the **rmic** compiler, see the RMIC tool page:

- Solaris, Linux, AIX, and z/OS version: http://download.oracle.com/javase/7/docs/technotes/tools/solaris/rmic.html
- Windows version: http://download.oracle.com/javase/7/docs/technotes/tools/windows/rmic.html

The idlj compiler:

Reference information for the **idlj** compiler.

Purpose

The **idlj** compiler generates Java bindings from an IDL file. This compiler supports the CORBA Objects By Value feature, which is required for inter-operation with RMI-IIOP. It is written in Java, and so can run on any platform.

More Information

To learn more about using the **idlj** compiler, see IDL-to-Java Compiler User's Guide.

Making RMI programs use IIOP:

A general guide to converting an RMI application to use RMI-IIOP.

Before you begin

To use these instructions, your application must already use RMI.

Procedure

1. If you are using the RMI registry for naming services, you must switch to CosNaming:
   a. In both your client and server code, create an InitialContext for JNDI. For a Java application use the following code:

      ```
      import javax.naming.*;
      ...
      Context ic = new InitialContext();
      ```

      For an applet, use this alternative code:

      ```
      import java.util.*;
      import javax.naming.*;
      ...
      Hashtable env = new Hashtable();
      env.put("java.naming.applet", this);
      Context ic = new InitialContext(env);
      ```

   b. Modify all uses of RMI registry lookup(), bind(), and rebind() to use JNDI lookup(), bind(), and rebind() instead. Instead of:

      ```
      import java.rmi.*;
      ...
      Naming.rebind("MyObject", myObj);
      ```

      use:

      ```
      import javax.naming.*;
      ...
      ic.rebind("MyObject", myObj);
      ```

2. If you are not using the RMI registry for naming services, you must have some other way of bootstrapping your initial remote object reference. For example, your server code might be using Java serialization to write an RMI object reference to an ObjectOutputStream and passing this to your client code for deserializing into an RMI stub. When doing this in RMI-IIOP, you must also ensure that object references are connected to an ORB before serialization and after deserialization.

a. On the server side, use the PortableRemoteObject.toStub() call to obtain a stub, then use writeObject() to serialize this stub to an ObjectOutputStream. If necessary, use Stub.connect() to connect the stub to an ORB before serializing it. For example:

```
org.omg.CORBA.ORB myORB = org.omg.CORBA.ORB.init(new String[0], null);
Wombat myWombat = new WombatImpl();
javax.rmi.CORBA.Stub myStub = (javax.rmi.CORBA.Stub)PortableRemoteObject.toStub(myWombat);
myStub.connect(myORB);
// myWombat is now connected to myORB.  To connect other objects to the
// same ORB, use PortableRemoteObject.connect(nextWombat, myWombat);
FileOutputStream myFile = new FileOutputStream("t.tmp");
ObjectOutputStream myStream = new ObjectOutputStream(myFile);
myStream.writeObject(myStub);
```

b. On the client side, use readObject() to deserialize a remote reference to the object from an ObjectInputStream. Before using the deserialized stub to call remote methods, it must be connected to an ORB. For example:

```
FileInputStream myFile = new FileInputStream("t.tmp");
ObjectInputStream myStream = new ObjectInputStream(myFile);
Wombat myWombat = (Wombat)myStream.readObject();
org.omg.CORBA.ORB myORB = org.omg.CORBA.ORB.init(new String[0], null);
((javax.rmi.CORBA.Stub)myWombat).connect(myORB);
// myWombat is now connected to myORB.  To connect other objects to the
// same ORB, use PortableRemoteObject.connect(nextWombat, myWombat);
```

The JNDI approach is much simpler, so it is preferable to use it whenever possible.

3. Either change your remote implementation classes to inherit from javax.rmi.PortableRemoteObject, or explicitly to export implementation objects after creation by calling PortableRemoteObject.exportObject(). For more discussion on this topic, read "Connecting IIOP stubs to the ORB" on page 138.
4. Change all the places in your code where there is a Java cast of a remote interface to use javax.rmi.PortableRemoteObject.narrow().
5. Do not depend on distributed garbage collection (DGC) or use any of the RMI DGC facilities. Use PortableRemoteObject.unexportObject() to make the ORB release its references to an exported object that is no longer in use.
6. Regenerate the RMI stubs and ties using the **rmic** command with the **-iiop** option. This will produce stub and tie files with the following names:

   _<*implementationName*>_Tie.class
   _<*interfaceName*>_Stub.class
7. Before starting the server, start the CosNaming server (in its own process) using the tnameserv command The CosNaming server uses the default port number of 2809. If you want to use a different port number, use the **-ORBInitialPort** parameter.
8. When starting client and server applications, you must specify some system properties. When running an application, you can specify properties on the command line:

   ```
   java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
        -Djava.naming.provider.url=iiop://<hostname>:2809
        <appl_class>
   ```
9. If the client is an applet, you must specify some properties in the applet tag. For example:

   ```
   java.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
   java.naming.provider.url=iiop://<hostname>:2809
   ```

   This example uses the default name service port number of 2809. If you specify a different port in the previous step, you need to use the same port number in

the provider URL here. The <*hostname*> in the provider URL is the host name that was used to start the CosNaming server.

Results

Your application can now communicate with CORBA objects using RMI-IIOP.

Connecting IIOP stubs to the ORB:

When your application uses IIOP stubs, as opposed to JRMP stubs, you must properly connect the IIOP stubs with the ORB before starting operations on the IIOP stubs (this is not necessary with JRMP stubs). This section discusses the extra 'connect' step required for the IIOP stub case.

The PortableRemoteObject.exportObject() call only creates a Tie object and caches it for future usage. The created tie does not have a delegate or an ORB associated. This is known as explicit invocation.

The PortableRemoteObject.exportObject() happens automatically when the servant instance is created. The servant instance is created when a PortableRemoteObject constructor is called as a base class. This is known as implicit invocation.

Later, when the application calls PortableRemoteObject.toStub(), the ORB creates the corresponding Stub object and associates it with the cached Tie object. But because the Tie is not connected and does not have a delegate, the newly created Stub also does not have a delegate or ORB.

The delegate is set for the stub only when the application calls Stub.connect(orb). Thus, any operations on the stub made before the ORB connection is made will fail.

The Java Language to OMG IDL Mapping Specification (http://www.omg.org/cgi-bin/doc?formal/01-06-07) says this about the Stub.connect() method:

> "The connect method makes the stub ready for remote communication using the specified ORB object orb. Connection normally happens implicitly when the stub is received or sent as an argument on a remote method call, but it is sometimes useful to do this by making an explicit call (e.g., following deserialization). If the stub is already connected to orb (has a delegate set for orb), then connect takes no action. If the stub is connected to some other ORB, then a RemoteException is thrown. Otherwise, a delegate is created for this stub and the ORB object orb."

For servants that are not POA-activated, Stub.connect(orb) is necessary as a required setup.

Restrictions when running RMI programs over IIOP:

A list of limitations when running RMI programs over IIOP.

To make existing RMI programs run over IIOP, observe the following restrictions.

- Make sure all constant definitions in remote interfaces are of primitive types or String and evaluated at compile time.

- Do not use Java names that conflict with IDL mangled names generated by the Java-to-IDL mapping rules. See section 28.3.2 of the Java Language to OMG IDL Mapping Specification for more information: http://www.omg.org/cgi-bin/doc?formal/01-06-07
- Do not inherit the same method name into a remote interface more than once from different base remote interfaces.
- Be careful when using names that are identical other than their case. The use of a type name and a variable of that type with a name that differs from the type name in case only is supported. Most other combinations of names that are identical other than their case are not supported.
- Do not depend on run time sharing of object references to be preserved exactly when transmitting object references to IIOP. Runtime sharing of other objects is preserved correctly.
- Do not use the following features of RMI, which do not work in RMI-IIOP:
  - RMISocketFactory
  - UnicastRemoteObject
  - Unreferenced
  - The Distributed Garbage Collector (DGC) interfaces

Additional information

Information about thread safety, working with other ORBs, the difference between UnicastRemoteObject and PortableRemoteObject, and known limitations.

Servers must be thread safe

Because remote method invocations on the same remote object might execute concurrently, a remote object implementation must be thread-safe.

Interoperating with other ORBs

RMI-IIOP should interoperate with other ORBs that support the CORBA 2.3 specification. It will not interoperate with older ORBs, because older ORBs cannot handle the IIOP encodings for Objects By Value. This support is needed to send RMI value classes (including strings) over IIOP.

Note: Although ORBs written in different languages should be able to interoperate, the Java ORB has not been fully tested with other vendors' ORBs.

When do I use UnicastRemoteObject vs PortableRemoteObject?

Use UnicastRemoteObject as the superclass for the object implementation in RMI programming. Use PortableRemoteObject in RMI-IIOP programming. If PortableRemoteObject is used, you can switch the transport protocol to either JRMP or IIOP during run time.

Known limitations

- JNDI 1.1 does not support *java.naming.factory.initial*=com.sun.jndi.cosnaming.CNCtxFactory as an Applet parameter. Instead, it must be explicitly passed as a property to the InitialContext constructor. This capability is supported in JNDI 1.2.
- When running the Naming Service on Unix based platforms, you must use a port number greater than 1024. The default port is 2809, so this should not be a problem.

Implementing the Connection Handler Pool for RMI

Thread pooling for RMI Connection Handlers is not enabled by default.

About this task

To enable the connection pooling implemented at the RMI TCPTransport level, set the option

```
-Dsun.rmi.transport.tcp.connectionPool=true
```

This version of the Runtime Environment does not have a setting that you can use to limit the number of threads in the connection pool.

Enhanced BigDecimal

From Java 5.0, the IBM BigDecimal class has been adopted by Oracle as java.math.BigDecimal. The com.ibm.math.BigDecimal class is reserved for possible future use by IBM and is currently deprecated. Migrate existing Java code to use java.math.BigDecimal.

The new java.math.BigDecimal uses the same methods as both the previous java.math.BigDecimal and com.ibm.math.BigDecimal. Existing code using java.math.BigDecimal continues to work correctly. The two classes do not serialize.

To migrate existing Java code to use the java.math.BigDecimal class, change the import statement at the start of your `.java` file from: `import com.ibm.math.*;` to `import java.math.*;`.

Working in a multiple network stack environment

In a multiple network stack environment (CINET), when one of the stacks fails, no notification or Java exception occurs for a Java program that is listening on an INADDR_ANY socket. Also, when new stacks become available, the Java application does not become aware of them until it rebinds the INADDR socket.

To avoid this situation, when a TCP/IP stack comes online:

- If the **ibm.socketserver.recover** property is set to false (which is the default), an exception (NetworkRecycledException) is thrown to the application to allow it either to fail or to attempt to rebind.
- If the **ibm.socketserver.recover** property is set to true, Java attempts to redrive the socket connection on the new stack if listening on all addresses (addrs). If the socket bind cannot be replayed at that time, an exception (NetworkRecycledException) is thrown to the application to allow it either to fail or to attempt to rebind.

Both ServerSocket.accept() and ServerSocketChannel.accept() can throw NetworkRecycledException.

While a socket is listening for new connections, it maintains a queue of incoming connections. When NetworkRecycledException is thrown and the system attempts to rebind the socket, the connection queue is reset and connection requests in this queue are dropped.

Support for XToolkit

XToolkit is included by default. You need XToolkit when using the SWT_AWT bridge in Eclipse to build an application that uses both SWT and Swing.

Restriction: Motif is no longer supported and will be removed in a later release.

Related links:

- An example of integrating Swing into Eclipse RCPs: http://eclipsezone.com/eclipse/forums/t45697.html
- Reference Information in the Eclipse information center: http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/awt/SWT_AWT.html
- Set up information is available on the Oracle Corporation Web site: http://download.oracle.com/javase/7/docs/technotes/guides/awt/1.5/xawt.html

Support for the Java Attach API

Your application can connect to another "target" virtual machine using the Java Attach API. Your application can then load an agent application into the target virtual machine, for example to perform tasks such as monitoring status.

Code for agent applications, such as JMX agents or JVMTI agents, is normally loaded during virtual machine startup by specifying special startup parameters. Requiring startup parameters might not be convenient for using agents on applications that are already running, such as WebSphere Application Servers. You can use the Java Attach API to load an agent at any time, by specifying the process ID of the target virtual machine. The Attach API capability is sometimes called the "late attach" capability.

Support for the Attach API is disabled by default.Support for the Attach API is enabled by default for Java 7 SR 2 and earlier. To enhance security, support for the Attach API is disabled by default for Java 7 SR 3 and later. On z/OS systems, processes that use the default z/OS OMVS segment cannot enable the attach API for security reasons.

Security considerations

Security for the Java Attach API is handled by POSIX file permissions. On z/OS, you must use UNIX user permissions to protect your applications. It is not sufficient to rely on RACF® or system level security to protect your applications. The reason is that these mechanisms do not have the necessary UNIX permissions set up and configured for the Java Attach API to remain secure.

The Java Attach API creates files and directories in a common directory.

The key security features of the Java Attach API are:

- A process using the Java Attach API must be owned by the same UNIX user ID as the target process. This constraint ensures that only the target process owner or root can attach other applications to the target process.
- The common directory uses the sticky bit to prevent a user from deleting or replacing a subdirectory belonging to another user. To preserve the security of this mechanism, set the ownership of the common directory to ROOT. This directory will contain files such as `_attachlock`, `_master`, and `_notifier`, which

are used only for synchronization. These files can be owned by any user, and must have read and write permission. However, you can remove execute permission on these files, if present. The files are empty and will be recreated automatically if deleted.

- The files in the subdirectory for a process, with the exception of a lock file, are accessible only by the owner of a process. The subdirectory has owner read, write, and execute permissions plus group and world execute permissions. In this directory, read and write access are restricted to the owner only, except for the `attachNotificationSync` file, which must have world and group write permissions. This exception does not affect security because the file is used exclusively for synchronization and is never written to or read.
- Information about the target process can be written and read only by the owner.
- Java 5 SR10 allowed users in the same group to access to each others' processes. This capability was removed in later versions.

You must secure access to the Java Attach API capability to ensure that only authorized users or processes can connect to another virtual machine. If you do not intend to use the Java Attach API capability, disable this feature using a Java system property. Set the **com.ibm.tools.attach.enable** system property to the value **no**; for example:

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

The Attach API can be enabled by setting the **com.ibm.tools.attach.enable** system property to the value **yes**; for example:

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

Using the Java Attach API

By default, the target virtual machine is identified by its process ID. To use a different target, change the system property **com.ibm.tools.attach.id**; for example:

```
-Dcom.ibm.tools.attach.id=<process_ID>
```

The target process also has a human-readable “display name”. By default, the display name is the command line used to start Java. To change the default display name, use the **com.ibm.tools.attach.displayName** system property. The ID and display name cannot be changed after the application has started.

The Attach API creates working files in a common directory, which by default is called `.com_ibm_tools_attach` and is created in the system temporary directory. The system property **java.io.tmpdir** holds the value of the system temporary directory. On non-Windows systems, the system temporary directory is typically `/tmp`.

You can specify a different common directory from the default, by using the following Java system property:

```
-Dcom.ibm.tools.attach.directory=directory_name
```

This system property causes the specified directory, *directory_name*, to be used as the common directory. If the directory does not already exist, it is created, however the parent directory must already exist. For example, the following system property creates a common directory called `myattachapidir` in the `usr` directory. The `usr` directory must already exist.

```
-Dcom.ibm.tools.attach.directory=/usr/myattachapidir
```

The common directory must be located on a local drive; specifying a network mounted file system might result in incorrect behavior.

If your Java application ends abnormally, for example, following a crash or a SIGKILL signal, the process subdirectory is not deleted. The Java VM detects and removes obsolete subdirectories where possible. The subdirectory can also be deleted by the owning user ID.

On heavily loaded system, applications might experience timeouts when attempting to connect to target applications. The default timeout is 120 seconds. Use the **com.ibm.tools.attach.timeout** system property to specify a different timeout value in milliseconds. For example, to timeout after 60 seconds:

```
-Dcom.ibm.tools.attach.timeout=60000
```

A timeout value of zero indicates an indefinite wait.

For JMX applications, you can disable authentication by editing the `<JAVA_HOME>/jre/lib/management/management.properties` file. Set the following properties to disable authentication in JMX:

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

Problems with the Attach API result in one of the following exceptions:

- `com.ibm.tools.attach.AgentLoadException`
- `com.ibm.tools.attach.AgentInitializationException`
- `com.ibm.tools.attach.AgentNotSupportedException`
- `java.io.IOException`

A useful reference for information about the Attach API can be found at http://download.oracle.com/javase/7/docs/technotes/guides/attach/index.html. The IBM implementation of the Attach API is equivalent to the Oracle Corporation implementation. However, the IBM implementation cannot be used to attach to, or accept attach requests from, non-IBM virtual machines. To use the attach API to attach to target processes from your application, you must add the "tools.jar" library to the application classpath. This library is not required for the target processes to accept attach requests.

Chapter 6. Running Java applications

Java applications can be started using the **java** launcher or through JNI. Settings are passed to a Java application using command-line arguments, environment variables, and properties files.

The `java` and `javaw` commands

An overview of the **java** and **javaw** commands.

Purpose

The **java** and **javaw** tools start a Java application by starting a Java Runtime Environment and loading a specified class.

The **javaw** command is identical to **java**, and is supported on z/OS for compatibility with other platforms.

Usage

The JVM searches for the initial class (and other classes that are used) in three sets of locations: the bootstrap class path, the installed extensions, and the user class path. The arguments that you specify after the class name or `.jar` file name are passed to the main function.

The **java** and **javaw** commands have the following syntax:

```
java [ options ] <class> [ arguments ... ]
java [ options ] -jar <file.jar> [ arguments ... ]
javaw [ options ] <class> [ arguments ... ]
javaw [ options ] -jar <file.jar> [ arguments ... ]
```

Parameters

[*options***]**
Command-line options to be passed to the runtime environment.

<class>
Startup class. The class must contain a main() method.

<file.jar>
Name of the `.jar` file to start. It is used only with the **-jar** option. The named `.jar` file must contain class and resource files for the application, with the startup class indicated by the Main-Class manifest header.

[*arguments ...***]**
Command-line arguments to be passed to the main() function of the startup class.

Obtaining version information

You obtain the IBM build and version number for your Java installation using the **-version** or **-fullversion** options. You can also obtain version information for all `jar` files on the class path by using the **-Xjarversion** option.

Procedure

1. Open a shell prompt.
2. Type the following command:

   ```
   java -version
   ```

   You will see information similar to:

```
java version "1.7.0"
Java(TM) SE Runtime Environment (build pxi3270sr1-20120201_02(SR1))
IBM J9 VM (build 2.6, JRE 1.7.0 Linux x86-32 20120131_101270 (JIT enabled, AOT enabled)
J9VM - R26_JVM_26_20120125_1726_B100726
JIT - r11_20120130_22318
GC - R26_JVM_26_20120125_1044_B100654
J9CL - 20120131_101270)
JCL - 20120127_01 based on Oracle 7u3-b02
```

   Exact build dates and versions will change.
3. To obtain only the build information for the JVM, type the following command:

   ```
   java -fullversion
   ```

   You will see information similar to:

   ```
   java full version "JRE 1.7.0 IBM Windows 32 build pwi3270sr1-20120412_01 (SR1)"
   ```

What to do next

You can also list the version information for all available `jar` files on the class path, the boot class path, and in the extensions directory. Type the following command:

```
java -Xjarversion -version
```

You will see information similar to:

```
java version "1.7.0"
Java(TM) SE Runtime Environment (build pxi3270sr1-20120201_02(SR1))
IBM J9 VM (build 2.6, JRE 1.7.0 Linux x86-32 20120131_101270 (JIT enabled, AOT enabled)
J9VM - R26_JVM_26_20120125_1726_B100726
JIT - r11_20120130_22318
GC - R26_JVM_26_20120125_1044_B100654
J9CL - 20120131_101270)
JCL - 20120127_01 based on Oracle 7u3-b02
/opt/ibm/java-i386-70/jre/lib/i386/default/jclSC170/vm.jar VERSION: 2.6 (01-31-2012)
/opt/ibm/java-i386-70/jre/lib/se-service.jar
/opt/ibm/java-i386-70/jre/lib/math.jar
/opt/ibm/java-i386-70/jre/lib/jlm.jar
/opt/ibm/java-i386-70/jre/lib/ibmorb.jar
/opt/ibm/java-i386-70/jre/lib/ibmorbapi.jar
/opt/ibm/java-i386-70/jre/lib/ibmcfw.jar VERSION: CCX.CF [o1103.02]
...
```

The information available varies for each `jar` file and is taken from the **`Implementation-Version`** and **`Build-Level`** properties in the manifest of the `jar` file.

Specifying Java options and system properties

You can specify Java options and system properties directly on the command line. You can also use an options file or an environment variable.

About this task

The sequence of the Java options on the command line defines which options take precedence during startup. Rightmost options have precedence over leftmost options. In the following example, the **-Xjit** option takes precedence:

```
java -Xint -Xjit myClass
```

Use one of more of the options that are shown in the procedure to customize your runtime environment.

Procedure

1. Specify options or system properties on the command line. For example:
   ```
   java -Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump MyJavaClass
   ```
2. Create an environment variable that is called **IBM_JAVA_OPTIONS** containing the options. For example:
   ```
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait
   -Xdisablejavadump"
   ```
3. Create a file that contains the options, and specify that file on the command line or in the **IBM_JAVA_OPTIONS** environment variable by using the **-Xoptionsfile** parameter. For more information about constructing this file, see "-Xoptionsfile" on page 438.

Standard options

The definitions for the standard options.

See "JVM command-line options" on page 428 for information about nonstandard (-X) options.

-agentlib:<*libname*>**[=**<*options*>**]**
Loads a native agent library <*libname*>; for example **-agentlib:hprof**. For more information, specify **-agentlib:jdwp=help** and **-agentlib:hprof=help** on the command line.

-agentpath:*libname***[=**<*options*>**]**
Loads a native agent library by full path name.

-cp <*directories and .zip or .jar files separated by* **:**>
Sets the search path for application classes and resources. If **-classpath** and **-cp** are not used and the **CLASSPATH** environment variable is not set, the user class path is, by default, the current directory (.).

-classpath <*directories and .zip or .jar files separated by* **:**>
Sets the search path for application classes and resources. If **-classpath** and **-cp** are not used and the **CLASSPATH** environment variable is not set, the user class path is, by default, the current directory (.).

-D<*property name*>**=**<*value*>
Sets a system property.

-help or -?
Prints a usage message.

-javaagent:<*jarpath*>**[=**<*options*>**]**
Load a Java programming language agent. For more information, see the java.lang.instrument API documentation.

-jre-restrict-search
Include user private JREs in the version search.

-no-jre-restrict-search
Exclude user private JREs in the version search.

-showversion
Prints product version and continues.

-verbose:<*option*>**[,**<*option*>**...]**
Enables verbose output. Separate multiple options using commas. The available options are:

class
Writes an entry to stderr for each class that is loaded.

gc Writes verbose garbage collection information to stderr. Use **-Xverbosegclog** (see "Garbage Collector command-line options" on page 453 for more information) to control the output. See Verbose garbage collection logging"Verbose garbage collection logging" on page 334 for more information.

jni
Writes information to stderr describing the JNI services called by the application and JVM.

sizes
Writes information to stderr describing the active memory usage settings.

stack
Writes information to stderr describing the Java and C stack usage for each thread.

-version
Prints product version.

-version:<*value*>
Requires the specified version to run, for example "1.5".

-X Prints help on nonstandard options.

Globalization of the java command

The **java** and **javaw** launchers accept arguments and class names containing any character that is in the character set of the current locale. You can also specify any Unicode character in the class name and arguments by using Java escape sequences.

To do this, use the **-Xargencoding** command-line option.

-Xargencoding
Use argument encoding. To specify a Unicode character, use escape sequences in the form \u####, where # is a hexadecimal digit (0 to 9, A to F).

-Xargencoding:utf8
Use UTF8 encoding.

-Xargencoding:latin
Use ISO8859_1 encoding.

For example, to specify a class called HelloWorld using Unicode encoding for both capital letters, use this command:

```
java -Xargencoding '\u0048ello\u0057orld'
```

The **java** and **javaw** commands provide translated messages. These messages differ based on the locale in which Java is running. The detailed error descriptions and other debug information that is returned by **java** is in English.

The Just-In-Time (JIT) compiler

The IBM Just-In-Time (JIT) compiler dynamically generates machine code for frequently used bytecode sequences in Java applications and applets during their execution. The JIT compiler delivers new optimizations as a result of compiler research, improves optimizations implemented in previous versions of the JIT, and provides better hardware exploitation.

The JIT is included in both the IBM SDK and Runtime Environment, which is enabled by default in user applications and SDK tools. Typically, you do not start the JIT explicitly; the compilation of Java bytecode to machine code occurs transparently. You can disable the JIT to help isolate a problem. If a problem occurs when executing a Java application or an applet, you can disable the JIT to help isolate the problem. Disabling the JIT is a temporary measure only; the JIT is required to optimize performance.

For more information about the JIT, see JIT and AOT problem determination"JIT and AOT problem determination" on page 322.

Disabling the JIT

The JIT can be disabled in a number of different ways. Both command-line options override the **JAVA_COMPILER** environment variable.

About this task

Turning off the JIT is a temporary measure that can help isolate problems when debugging Java applications.

Procedure

- Set the **JAVA_COMPILER** environment variable to `NONE` or the empty string before running the **java** application. Type the following command at a shell prompt:

  ```
  export JAVA_COMPILER=NONE
  ```

- Use the **-D** option on the JVM command line to set the **java.compiler** property to `NONE` or the empty string. Type the following command at a shell prompt:

  ```
  java -Djava.compiler=NONE <class>
  ```

- Use the **-Xint** option on the JVM command line. Type the following command at a shell prompt:

  ```
  java -Xint <class>
  ```

Enabling the JIT

The JIT is enabled by default. You can explicitly enable the JIT in a number of different ways. Both command-line options override the **JAVA_COMPILER** environment variable.

Procedure

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

  ```
  export JAVA_COMPILER=jitc
  ```

If the **JAVA_COMPILER** environment variable is an empty string, the JIT remains disabled. To disable the environment variable, at the prompt, enter:

```
unset JAVA_COMPILER
```

- Use the **-D** option on the JVM command line to set the **java.compiler** property to `jitc`. At a prompt, enter:

  ```
  java -Djava.compiler=jitc <class>
  ```

- Use the **-Xjit** option on the JVM command line. Do **not** specify the **-Xint** option at the same time. At a prompt, enter:

  ```
  java -Xjit <class>
  ```

Determining whether the JIT is enabled

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

Procedure

Run the **java** launcher with the **-version** option. Enter the following command at a shell prompt:

```
java -version
```

If the JIT is not in use, a message is displayed that includes the following text:

```
(JIT disabled)
```

If the JIT is in use, a message is displayed that includes the following text:

```
(JIT enabled)
```

What to do next

For more information about the JIT, see The JIT compiler"The JIT compiler" on page 57.

Specifying a garbage collection policy

The Garbage Collector manages the memory used by Java and by applications running in the JVM.

When the Garbage Collector receives a request for storage, unused memory in the heap is set aside in a process called "allocation". The Garbage Collector also checks for areas of memory that are no longer referenced, and releases them for reuse. This is known as "collection".

The collection phase can be triggered by a memory allocation fault, which occurs when no space remains for a storage request, or by an explicit System.gc() call.

Garbage collection can significantly affect application performance, so the IBM virtual machine provides various methods of optimizing the way garbage collection is carried out, potentially reducing the effect on your application.

For more detailed information about garbage collection, see Detailed description of garbage collection"Detailed description of global garbage collection" on page 29.

Garbage collection options

The **-Xgcpolicy** options control the behavior of the Garbage Collector. They make trade-offs between throughput of the application and overall system, and the pause times that are caused by garbage collection.

The format of the option is as follows:

-Xgcpolicy:<value>

The following values are available:

gencon
The **gencon** policy (default) uses a concurrent mark phase combined with generational garbage collection to help minimize the time that is spent in any garbage collection pause. This policy is particularly useful for applications with many short-lived objects, such as transactional applications. Pause times can be significantly shorter than with the **optthruput** policy, while still producing good throughput. Heap fragmentation is also reduced.

balanced
The **balanced** policy uses mark, sweep, compact and generational style garbage collection. The concurrent mark phase is disabled; concurrent garbage collection technology is used, but not in the way that concurrent mark is implemented for other policies. The **balanced** 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 tries to avoid global collections by matching object allocation and survival rates. If you have problems with application pause times that are caused by global garbage collections, particularly compactions, this policy might improve application performance. For more information about this policy, including when to use it, see "Balanced Garbage Collection policy" on page 39.

optavgpause
The **optavgpause** policy uses concurrent mark and concurrent sweep phases. Pause times are shorter than with **optthruput**, but application throughput is reduced because some garbage collection work is taking place while the application is running. Consider using this policy if you have a large heap size (available on 64-bit platforms), because this policy limits the effect of increasing heap size on the length of the garbage collection pause. However, if your application uses many short-lived objects, the **gencon** policy might produce better performance.

subpool
The **subpool** policy is deprecated and is now an alias for **optthruput**. Therefore, if you use this option, the effect is the same as **optthruput**.

optthruput
The **optthruput** policy disables the concurrent mark phase. The application stops during global garbage collection, so long pauses can occur. This configuration is typically used for large-heap applications when high application throughput, rather than short garbage collection pauses, is the main performance goal. If your application cannot tolerate long garbage collection pauses, consider using another policy, such as **gencon**.

More effective heap usage using compressed references

Many Java application workloads depend on the Java heap size. The IBM SDK for Java can use compressed references on 64-bit platforms to decrease the size of Java

objects and make more effective use of the available space. The result is less frequent garbage collection and improved memory cache utilization.

If you specify the **-Xnocompressedrefs** command-line option, the IBM SDK for Java 64-bit stores object references as 64-bit values. If you specify the **-Xcompressedrefs** command-line option, object references are stored as 32-bit representation, which reduces the 64-bit object size to be the same as a 32-bit object.

As the 64-bit objects with compressed references are smaller than default 64-bit objects, they occupy a smaller memory footprint in the Java heap and improves data locality. This results in better memory utilization and improved performance.

If you are using a 64-bit IBM SDK for Java, use **-Xcompressedrefs** whenever you require a maximum heap size of less than 25 GB. For example, your application might use a lot of native memory and require the JVM to run in a small footprint.

Note: If you are using compressed references on z/OS v1.10 or earlier, you must use APAR OA26294.

See "Compressed references" on page 27 for more detailed information and hardware/operating system specific guidance on compressed references. More information is also available in the Websphere white paper on compressed references.

Pause time

If an object cannot be created from the available space in the heap, the Garbage Collector attempts to tidy the heap. The intention is that subsequent allocation requests can be satisfied quickly.

The Garbage Collector tries to returning the heap to a state in which the immediate and subsequent space requests are successful. The Garbage Collector identifies unreferenced "garbage" objects, and deletes them. This work takes place in a garbage collection cycle. These cycles might introduce occasional, unexpected pauses in the execution of application code. As applications grow in size and complexity, and heaps become correspondingly larger, the garbage collection pause time tends to grow in size and significance. Pause time can vary from a few milliseconds to many seconds. The actual time depends on the size of the heap, and the quantity of garbage.

The **-Xgcpolicy:balanced** command-line option uses a garbage collection policy with reduced pause times, even as the Java heap size grows.

Pause time reduction

The JVM uses multiple techniques to reduce pause times, including concurrent garbage collection, partial garbage collection, and generational garbage collection.

The **-Xgcpolicy:optavgpause** command-line option requests the use of concurrent garbage collection (GC) to reduce significantly the time that is spent in garbage collection pauses. Concurrent GC reduces the pause time by performing some garbage collection activities concurrently with normal program execution to minimize the disruption caused by the collection of the heap. The **-Xgcpolicy:optavgpause** option also limits the effect of increasing the heap size on the length of the garbage collection pause. The **-Xgcpolicy:optavgpause** option is most useful for configurations that have large heaps. With the reduced pause time, you might experience some reduction of throughput to your applications.

During concurrent GC, a significant amount of time is wasted identifying relatively long-lasting objects that cannot then be collected. If garbage collection concentrates on only the objects that are most likely to be recyclable, you can further reduce pause times for some applications. Generational GC reduces pause times by dividing the heap into two generations: the "new" and the "tenure" areas. Objects are placed in one of these areas depending on their age. The new area is the smaller of the two and contains new objects; the tenure is larger and contains older objects. Objects are first allocated to the new area; if they have active references for long enough, they are promoted to the tenure area.

Generational GC depends on most objects not lasting long. Generational GC reduces pause times by concentrating the effort to reclaim storage on the new area because it has the most recyclable space. Rather than occasional but lengthy pause times to collect the entire heap, the new area is collected more frequently and, if the new area is small enough, pause times are comparatively short. However, generational GC has the drawback that, over time, the tenure area might become full. To minimize the pause time when this situation occurs, use a combination of concurrent GC and generational GC. The **-Xgcpolicy:gencon** option requests the combined use of concurrent and generational GC to help minimize the time that is spent in any garbage collection pause.

Partial GC is like generational GC because different areas of the Java heap are collected independently. In both cases, using more than one area in the Java heap allows garbage collection to occur more frequently, with shorter pause times. However, there are two important differences that make partial GC better than generational GC at avoiding long application pauses:

- There are many areas, called regions, defined in the Java heap instead of just the tenure and new area.
- Any number of these regions can be collected at the same time.

With generational GC, short pauses are possible while collecting only the new area. However, there is an inevitably long pause required to occasionally collect the tenure and new area together. The **-Xgcpolicy:balanced** option requests a combined use of concurrent and partial garbage collection.

Environments with very full heaps

If the Java heap becomes nearly full, and very little garbage can be reclaimed, requests for new objects might not be satisfied quickly because no space is immediately available.

If the heap is operated at near-full capacity, application performance might suffer regardless of which garbage collection options are used; and, if requests for more heap space continue to be made, the application might receive an OutOfMemoryError, which results in JVM termination if the exception is not caught and handled. At this point, the JVM produces a Javadump file for use during diagnostic procedures. In these conditions, you are recommended either to increase the heap size by using the **-Xmx** option or to reduce the number of objects in use.

For more information, see Garbage Collector diagnostic data"Garbage Collector diagnostic data" on page 333.

Euro symbol support

The IBM SDK and Runtime Environment set the Euro as the default currency for those countries in the European Monetary Union (EMU) for dates on or after 1 January, 2002. From 1 January 2008, Cyprus and Malta also have the Euro as the default currency.

To use the old national currency, specify **–Duser.variant=PREEURO** on the Java command line.

If you are running the UK, Danish, or Swedish locales and want to use the Euro, specify **–Duser.variant=EURO** on the Java command line.

Configuring large page memory allocation

You can enable large page support, on systems that support it, by starting Java with the **-Xlp** option. 1M pageable pages, when available, are the default size for the object heap and code cache.

About this task

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

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

Large page size	"-Xlp:codecache" on page 436	"-Xlp:objectheap" on page 436	"-Xlp" on page 435
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** options, see "JVM command-line options" on page 428.

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 or later 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 APAR OA41307, and the z/OS V1.13 Remote Storage Manager Enablement Offering web deliverable.
- From service refresh 4, 1M pageable, when available, is the default page size for the object heap and the code cache.

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 running on a system that has Flash cards installed, and using a z/OS system that supports Flash, 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 http://publib.boulder.ibm.com/infocenter/zos/v1r13/index.jsp?topic=%2Fcom.ibm.zos.r13.ieae100%2Flfarea.htm.

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.

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 "Garbage Collector command-line options" on page 453.

Chapter 7. Performance

You can improve the performance of applications by tuning the product, enabling hardware features, or using specific APIs in your application code.

Here are some methods for improving performance:

- Share class data between JVMs. For more information, see "Class data sharing between JVMs."
- Choose the best garbage collection policy for your application. For more information, see "Specifying a garbage collection policy" on page 150.
- Enable large page support on your operating system. For more information, see "Configuring large page memory allocation" on page 154.

Class data sharing between JVMs

Class data sharing enables multiple JVMs to share a single space in memory.

You can share class data between Java Virtual Machines (JVMs) by storing it in a cache in shared memory. Sharing reduces the overall virtual storage consumption when more than one JVM shares a cache. Sharing also reduces the startup time for a JVM after the cache has been created. The shared class cache is independent of any running JVM and persists until it is deleted or until a system IPL.

A shared cache can contain:

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

Overview of class data sharing

Class data sharing provides a method of reducing memory footprint and improving JVM start time.

Enabling class data sharing

Enable class data sharing by using the **-Xshareclasses** option when starting a JVM. The JVM connects to an existing cache or creates a new cache if one does not exist.

All bootstrap and application classes loaded by the JVM are shared by default. Custom class loaders share classes automatically if they extend the application class loader. Otherwise, they must use the Java Helper API provided with the JVM to access the cache. See "Adapting custom class loaders to share classes" on page 168.

The JVM can also store ahead-of-time (AOT) compiled code in the cache for certain methods to improve the startup time of subsequent JVMs. The AOT compiled code is not shared between JVMs, but is cached to reduce compilation time when the JVM starts. The amount of AOT code stored in the cache is determined heuristically. You cannot control which methods get stored in the cache. You can

set maximum and minimum limits on the amount of cache space used for AOT code, or you can disable AOT caching completely. See “Class data sharing command-line options” on page 159 for more information.

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 “Class data sharing command-line options” on page 159 for more information.

Cache access

A JVM can access a cache with either read/write or read-only access. Any JVM connected to a cache with read/write access can update the cache. Any number of JVMs can concurrently read from the cache, even while another JVM is writing to it.

You must take care if runtime bytecode modification is being used. See “Runtime bytecode modification” on page 167 for more information.

Dynamic updating of the cache

The shared class cache persists beyond the lifetime of any JVM. Therefore, the cache is updated dynamically to reflect any modifications that might have been made to JARs or classes on the file system. The dynamic updating makes the cache independent of the application using it.

Cache security

Access to the shared class cache is limited by operating system permissions and Java security permissions. The shared class cache is created with user access by default unless the **groupAccess** command-line suboption is used. Only a class loader that has registered to share class data can update the shared class cache.

A cache can be accessed only by a JVM running in the same storage key as the JVM that created the cache. If the keys do not match, permission to access the cache is denied. Known environments where storage keys can cause an issue include:

WebSphere control region (key 2)
Attempting to access the shared cache for the WebSphere control region generates the following error message:

```
JVMSHRC337W Platform error message: shmat : EDC5111I Permission denied.
```

CICS, when switching between STGPROT=NO (key 8) and STGPROT=YES (key 9)
If CICS is started with **STGPROT=YES**, CICS allocates the shared cache in key 9. This enables cache sharing between programs running in CICS key (8) and User key (9). If CICS is started with **STGPROT=NO**, the cache is allocated in key 8. Using a shared cache in key 8 might lead to errors if the CICS region is subsequently restarted with **STGPROT=YES**. Unless the cache is rebuilt, a program running in User key (9) will be unable to access the shared class cache. In this situation the JVM issues a message, similar to:

```
JVMSHRC337W Platform error message: shmat : EDC5111I Permission denied.
```

The default storage key for the JVM is key 8.

(31-bit only) The cache memory is protected against accidental or deliberate corruption using memory page protection. This protection is not an absolute guarantee against corruption because the JVM must unprotect pages to write to them. The only way to guarantee that a cache cannot be modified is to open it read-only.

(64-bit only) Memory page protection using `PGSER PROTECT` is unavailable on z/OS 64-bit mode.

If a Java SecurityManager is installed, classloaders, excluding the default bootstrap, application, and extension class loaders, must be granted permission to share classes. Grant permission by adding SharedClassPermission lines to the `java.policy` file. See "Using SharedClassPermission" on page 168 for more information. The RuntimePermission createClassLoader restricts the creation of new class loaders and therefore also restricts access to the cache.

Cache lifespan

Multiple caches can exist on a system and you specify them by name as a suboption to the **-Xshareclasses** command. A JVM can connect to only one cache at any one time.

You can override the default cache size on startup using **-Xscmx<n><size>**. This size is then fixed for the lifetime of the cache. Caches exist until they are explicitly deleted using a suboption to the **-Xshareclasses** command or until the next system IPL.

Cache utilities

All cache utilities are suboptions to the **-Xshareclasses** command. See "Class data sharing command-line options" or use **-Xshareclasses:help** to see a list of available suboptions.

Class data sharing command-line options

Class data sharing and the cache management utilities are controlled using command-line options to the Java launcher.

For options that take a <*size*> parameter, suffix the number with "k" or "K" to indicate kilobytes, "m" or "M" to indicate megabytes, or "g" or "G" to indicate gigabytes.

-Xscmaxaot<*size*>
: Sets the maximum number of bytes in the cache that can be used for AOT data. Use this option to ensure that a certain amount of cache space is available for non-AOT data. By default, the maximum limit for AOT data is the amount of free space in the cache. The value of this option should not be smaller than the value of **-Xscminaot** and must not be larger than the value of **-Xscmx**.

-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 that is 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**.

-Xscminaot<*size*>
Sets the minimum number of bytes in the cache to reserve for AOT data. By default, no space is reserved for AOT data, although AOT data is written to the cache until the cache is full or the **-Xscmaxaot** limit is reached. The value of this option must not exceed the value of **-Xscmx** or **-Xscmaxaot**. The value of **-Xscminaot** must always be considerably less than the total cache size because AOT data can be created only for cached classes. If the value of **-Xscminaot** is equal to the value of **-Xscmx**, no class data or AOT data is stored because AOT data must be associated with a class in the cache.

-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 **-Xscmaxjit** 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.

-Xscdmx<*size*>
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.

size can be a percentage, expressed as a number, or an absolute value.

A class debug area is still created if you use the **-Xnolinenumbers** option with the **-Xscdmx** option on the command line.

-Xscmx<*size*>
Specifies cache size. This option applies only if a cache is being created and no cache of the same name exists. The default cache size is platform-dependent. You can find out the size value being used by adding **-verbose:sizes** as a command-line argument. The minimum cache size is 4 KB. The maximum cache size is also platform-dependent. (See "Cache size limits" on page 166.)

-Xshareclasses:<*suboption*>**[,**<*suboption*>**...]**
Enables class data sharing. Can take a number of suboptions, some of which are cache utilities. Cache utilities perform the required operation on the specified cache, without starting the VM. You can combine multiple suboptions, separated by commas, but the cache utilities are mutually exclusive. When running cache utilities, the message `Could not create the Java virtual machine` is expected. Cache utilities do not create the virtual machine.

Some cache utilities can work with caches from previous Java versions or caches that are created by JVMs with different bit-widths. These caches are referred to as "incompatible" caches.

You can use the following suboptions with the **-Xshareclasses** option:

help
Lists all the command-line suboptions.

name=<*name*>
Connects to a cache of a given name, creating the cache if it does not already exist. Also used to indicate the cache that is to be modified by

cache utilities; for example, **destroy**. Use the **listAllCaches** utility to show which named caches are currently available. If you do not specify a name, the default name "sharedcc_%u" is used. %u in the cache name inserts the current user name. You can specify "%g" in the cache name to insert the current group name.

Note: Some features, when enabled, result in the creation of caches that cannot be shared with caches that were created when the feature was disabled. The multitenancy support is one such example. In this situation, you can have more than one cache with the same name.

cacheDir=<*directory*>
Sets the directory in which cache data is read and written. By default, <*directory*> is `/tmp/javasharedresources`. The user must have sufficient permissions in <*directory*>. Caches are stored in shared memory and have control files that describe the location of the memory. Control files are stored in a `javasharedresources` subdirectory of the **cacheDir** specified. Do not move or delete control files in this directory. The **listAllCaches** utility, the **destroyAll** utility, and the **expire** suboption work only in the scope of a given **cacheDir**.

cacheDirPerm=<*permission*>
Sets UNIX-style permissions when creating a cache directory. <*permission*> must be a number in the ranges 0700 - 0777 or 1700 - 1777. If <*permission*> is not valid, the JVM terminates with an appropriate error message.

The permissions that are 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`, the **cacheDirPerm** suboption is ignored and the cache directory permissions are set to 777.If you do not set the **cacheDirPerm** suboption, and the cache directory does not already exist, a new directory is created with permissions set to 777, for compatibility with earlier Java versions. Permissions for existing cache directories are unchanged, to avoid generating RACF errors, which generate log messages.

disableBCI
Turns off BCI support. This option can be used to override **-XX:ShareClassesEnableBCI**. For more information, see "-XX:ShareClassesEnableBCI" on page 447.

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 353. 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.

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.

readonly
Opens an existing cache with read-only permissions. The JVM does not create a new cache with this suboption. Opening a cache read-only prevents the JVM from making any updates to the cache. It also allows the JVM to connect to caches created by other users or groups without requiring write access. By default, this suboption is not specified.

groupAccess
Sets operating system permissions on a new cache to allow group access to the cache. Group access can be set only when permitted by the operating system **umask** setting. The default is user access only.

verbose
Enables verbose output, which provides overall status on the shared class cache and more detailed error messages.

verboseAOT
Enables verbose output when compiled AOT code is being found or stored in the cache. AOT code is generated heuristically. You might not see any AOT code generated at all for a small application. You can disable AOT caching by using the **noaot** suboption.

verboseIO
Gives detailed output on the cache I/O activity, listing information on classes that are stored and found. Each class loader is given a unique ID (the bootstrap loader is always 0) and the output shows the class loader hierarchy at work, where class loaders must ask their parents for a class before they can load it themselves. It is usual to see many failed requests; this behavior is expected for the class loader hierarchy.

verboseHelper
Enables verbose output for the Java Helper API. This output shows you how the Helper API is used by your class loader.

silent
Turns off all shared classes messages, including error messages. Unrecoverable error messages, which prevent the JVM from initializing, are displayed.

nonfatal
Allows the JVM to start even if class data sharing fails. Normal behavior for the JVM is to refuse to start if class data sharing fails. If you select **nonfatal** and the shared classes cache fails to initialize, the JVM attempts to connect to the cache in read-only mode. If this attempt fails, the JVM starts without class data sharing.

none
Can be added to the end of a command line to disable class data sharing. This suboption overrides class sharing arguments found earlier on the command line. 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 "Obtaining information about shared caches" on page 360.

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 "Obtaining information about shared caches" on page 360.

modified=<*modified context*>
Used when a JVMTI agent is installed that might modify bytecode at run time. If you do not specify this suboption and a bytecode modification agent is installed, classes are safely shared with an extra performance cost. The <*modified context*> is a descriptor that is chosen by the user; for example, "myModification1". This option partitions the cache, so that only JVMs that use context myModification1 can share the same classes. For instance, if you run HelloWorld with a modification context and then run it again with a different modification context, all classes are stored twice in the cache. For more information, see "Runtime bytecode modification" on page 167.

reset
Causes a cache to be destroyed and then recreated when the JVM starts up. Can be added to the end of a command line as **-Xshareclasses:reset**.

destroy (Utility option)
Destroys a cache specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. A cache can be destroyed only if all JVMs using it have shut down, and the user has sufficient permissions.

destroyAll (Utility option)
Tries to destroy all caches available using the specified **cacheDir** and **nonpersistent** suboptions. A cache can be destroyed only if all JVMs using it have shut down, and the user has sufficient permissions.

expire=<*time in minutes*>
Destroys all caches that have been unused for the time that is specified before loading shared classes. This option is not a utility option because it does not cause the JVM to exit.

listAllCaches (Utility option)
Lists all the compatible and incompatible caches that exist in the specified cache directory. If you do not specify **cacheDir**, the default directory is used. Summary information, such as Java version and current usage is displayed for each cache.

Note: Some features, when enabled, result in the creation of caches that cannot be shared with caches that are created when the feature is disabled. The multitenancy support is one such example. In this situation, you can have more than one cache with the same name. The output from the **listAllCaches** option has a `feature` column which lists the feature that created the cache, usually `default`. For multitenancy support, the feature is `mt`, and the cache is listed in the `Incompatible shared caches` section of the output.

printStats[=<*data_types*>**] (Utility option)**
Displays summary information for the cache that is specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. The most useful information that is displayed is how full the cache is and how many classes it contains. Stale classes are classes that are updated on the file system and which the cache has therefore marked as "stale". Stale classes are not purged from the cache and can be reused.

Specify one or more data types, which are 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.For more information, see “printStats utility” on page 362.

printAllStats (Utility option)
Displays detailed information for the cache that is specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. Every class is listed in chronological order, with a reference to the location from which it was loaded. AOT code for class methods is also listed.

For more information, see “printAllStats utility” on page 366.

(31-bit only) mprotect=[all | default | none]
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 small performance cost. Specifying `all` ensures that all the cache pages are protected, including the header. Specifying `none` disables the page protection.

noBootclasspath
Prevents storage of classes that are loaded by the bootstrap class loader in the shared classes cache. Can be used with the SharedClassURLFilter API to control exactly which classes get cached. For more information about shared class filtering, see “Using the SharedClassHelper API” on page 359.

cacheRetransformed
Enables caching of classes that are transformed by using the JVMTI RetransformClasses function.

noaot
Disables caching of AOT code. AOT code already in the shared data cache can be loaded.

nojitdata
Disables caching of JIT data. JIT data already in the shared data cache can be loaded.

-Xzero<*options*>
Enables reduction of the memory footprint of Java when concurrently running multiple Java invocations. **-Xzero** might not be appropriate for all types of applications because it changes the implementation of `java.util.ZipFile`, which might cause extra memory usage.

The <*options*> *sharezip* and *sharebootzip* apply to class data sharing. The option *sharebootzip* is enabled by default and can be turned off by using **-Xzero:nosharebootzip**. For more information about **-Xzero** and the options available, see "JVM command-line options" on page 428.

Creating, populating, monitoring, and deleting a cache

An overview of the life-cycle of a shared class data cache including examples of the cache management utilities.

To enable class data sharing, add **-Xshareclasses[:name=<name>]** to your application command line.

The JVM either connects to an existing cache of the given name or creates a new cache of that name. If a new cache is created, it is populated with all bootstrap and application classes being loaded until the cache becomes full. If two or more JVMs are started concurrently, they populate the cache concurrently.

To check that the cache has been created, run `java -Xshareclasses:listAllCaches`. To see how many classes and how much class data is being shared, run `java -Xshareclasses:[name=<name>],printStats`. You can run these utilities after the application JVM has terminated or in another command window.

Note: Some features, when enabled, result in the creation of caches that cannot be shared with caches that are created when the feature is disabled. The multitenancy support is one such example. In this situation, you can have more than one cache with the same name. The output from the **listAllCaches** option has a `feature` column which lists the feature that created the cache, usually `default`. For multitenancy support, the feature is `mt`, and the cache is listed in the `Incompatible shared caches` section of the output.

For more feedback on cache usage while the JVM is running, use the **verbose** suboption. For example, `java -Xshareclasses:[name=<name>],verbose`.

To see classes being loaded from the cache or stored in the cache, add `-Xshareclasses:[name=<name>],verboseIO` to your application command line.

To delete the cache, run `java -Xshareclasses:[name=<name>],destroy`. You usually delete caches only if they contain many stale classes or if the cache is full and you want to create a bigger cache.

You should tune the cache size for your specific application, because the default is unlikely to be the optimum size. To determine the optimum cache size, specify a large cache, using **-Xscmx**, run the application, and then use **printStats** to

determine how much class data has been stored. Add a small amount to the value shown in **printStats** for contingency. Because classes can be loaded at any time during the lifetime of the JVM, it is best to do this analysis after the application has terminated. However, a full cache does not have a negative affect on the performance or capability of any JVMs connected to it, so it is acceptable to decide on a cache size that is smaller than required.

If a cache becomes full, a message is displayed on the command line of any JVMs using the **verbose** suboption. All JVMs sharing the full cache then loads any further classes into their own process memory. Classes in a full cache can still be shared, but a full cache is read-only and cannot be updated with new classes.

Performance and memory consumption

Class data sharing is particularly useful on systems that use more than one JVM running similar code; the system benefits from reduced real storage consumption. It is also useful on systems that frequently start and shut down JVMs, which benefit from the improvement in startup time.

The processor and memory usage required to create and populate a new cache is minimal. The JVM startup cost in time for a single JVM is typically between 0 and 5% slower compared with a system not using class data sharing, depending on how many classes are loaded. JVM startup time improvement with a populated cache is typically between 10% and 40% faster compared with a system not using class data sharing, depending on the operating system and the number of classes loaded. Multiple JVMs running concurrently show greater overall startup time benefits.

Duplicate classes are consolidated in the shared class cache. For example, class A loaded from `myClasses.jar` and class A loaded from `myOtherClasses.jar` (with identical content) is stored only once in the cache. The **printAllStats** utility shows multiple entries for duplicated classes, with each entry pointing to the same class.

When you run your application with class data sharing, you can use the operating system tools to see the reduction in virtual storage consumption.

Considerations and limitations of using class data sharing

Consider these factors when deploying class data sharing in a product and using class data sharing in a development environment.

Cache size limits

The maximum theoretical cache size is 2 GB. The size of cache you can specify is limited by the amount of physical memory and swap space available to the system.

Because the virtual address space of a process is shared between the shared classes cache and the Java heap, if you increase the maximum size of the Java heap you might reduce the size of the shared classes cache you can create.

JVMTI RetransformClasses() is unsupported

You cannot run RetransformClasses() on classes loaded from the shared class cache.

The JVM might throw the exception UnmodifiableClassException if you attempt to run RetransformClasses(). It does not work because class file bytes are not available for classes loaded from the shared class cache. If you must use

RetransformClasses(), ensure that the classes to be transformed are not loaded from the shared class cache, or disable the shared class cache feature.

Required APAR for Shared Classes

You must apply z/OS APAR OA11519, available for z/OS R1.6 and onwards, to any z/OS system where shared classes are used. This APAR ensures that multiple shmat requests for the same shared segment will map to the same virtual address for multiple JVMs.

Without this APAR, there is a problem with using shared memory when multiple JVMs are stored in a single address space. Each shmat call consumes a separate virtual address range. This is not acceptable because shared classes will run out of shared memory pages prematurely.

Working with BPXPRMxx settings

Some of the **BPXPRMxx** parmlib settings affect shared classes performance. Using the wrong settings can stop shared classes from working. These settings might also have performance implications.

For further information about performance implications, and use of these parameters, see the *z/OS MVS Initialization and Tuning Reference (SA22-7592)* at http://publibz.boulder.ibm.com/epubs/pdf/iea2e280.pdf and the *z/OS Unix System Services Planning Guide (GA22-7800)* at http://publibz.boulder.ibm.com/epubs/pdf/bpxzb280.pdf. The most significant **BPXPRMxx** parameters that affect the operation of shared classes are:

- **MAXSHAREPAGES**, **IPCSHMSPAGES**, **IPCSHMMPAGES**, and **IPCSHMNSEGS**. These settings affect the amount of shared memory pages available to the JVM. The JVM uses these memory pages for the shared classes cache. If you request large cache sizes, you might have to increase the amount of shared memory pages available.

  The shared page size for a z/OS UNIX System Service is fixed at 4 KB for 31-bit and 1 MB for 64-bit. Shared classes try to create a 16 MB cache by default on both 31- and 64-bit platforms. Therefore set **IPCSHMMPAGES** greater than 4096 on a 31-bit system.

  If you set a cache size with **-Xscmx**, the VM rounds up the value to the nearest megabyte. You must take this factoring into account when setting **IPCSHMMPAGES** on your system.
- **IPCSEMNIDS**, and **IPCSEMNSEMS**. These settings affect the amount of SystemV IPC semaphore available to UNIX processes. IBM shared classes use System V IPC semaphores to communicate between the JVMs.

Runtime bytecode modification

Any JVM using a JVM Tool Interface (JVMTI) agent that can modify bytecode data must use the **modified=**<*modified_context*> suboption if it wants to share the modified classes with another JVM.

The modified context is a user-specified descriptor that describes the type of modification being performed. The modified context partitions the cache so that all JVMs running under the same context share a partition.

This partitioning allows JVMs that are not using modified bytecode to safely share a cache with those that are using modified bytecode. All JVMs using a given modified context must modify bytecode in a predictable, repeatable manner for each class, so that the modified classes stored in the cache have the expected

modifications when they are loaded by another JVM. Any modification must be predictable because classes loaded from the shared class cache cannot be modified again by the agent.

If a JVMTI agent is used without a modification context, classes are still safely shared by the JVM, but with a small affect on performance. Using a modification context with a JVMTI agent avoids the need for extra checks and therefore has no affect on performance. A custom ClassLoader that extends java.net.URLClassLoader and modifies bytecode at load time without using JVMTI automatically stores that modified bytecode in the cache, but the cache does not treat the bytecode as modified. Any other VM sharing that cache loads the modified classes. You can use the **modified=**<*modification_context*> suboption in the same way as with JVMTI agents to partition modified bytecode in the cache. If a custom ClassLoader needs to make unpredictable load-time modifications to classes, that ClassLoader must not attempt to use class data sharing.

See "Dealing with runtime bytecode modification" on page 351 for more detail on this topic.

Operating system limitations

Temporary disk space must be available to hold cache information. The operating system enforces cache permissions.

The shared class cache requires disk space to store identification information about the caches that exist on the system. This information is stored in `/tmp/javasharedresources`. If the identification information directory is deleted, the JVM cannot identify the shared classes on the system and must re-create the cache. Use the **ipcs** command to view the memory segments used by a JVM or application.

Users running a JVM must be in the same group to use a shared class cache. The operating system enforces the permissions for accessing a shared class cache. If you do not specify a cache name, the user name is appended to the default name so that multiple users on the same system create their own caches by default.

Using SharedClassPermission

If a SecurityManager is being used with class data sharing and the running application uses its own class loaders, you must grant these class loaders shared class permissions before they can share classes.

You add shared class permissions to the `java.policy` file using the ClassLoader class name (wildcards are permitted) and either "read", "write", or "read,write" to determine the access granted. For example:

```
permission com.ibm.oti.shared.SharedClassPermission
        "com.abc.customclassloaders.*", "read,write";
```

If a ClassLoader does not have the correct permissions, it is prevented from sharing classes. You cannot change the permissions of the default bootstrap, application, or extension class loaders.

Adapting custom class loaders to share classes

Any class loader that extends java.net.URLClassLoader can share classes without modification. You must adopt class loaders that do not extend java.net.URLClassLoader to share class data.

You must grant all custom class loaders shared class permissions if a SecurityManager is being used; see “Using SharedClassPermission” on page 168. IBM provides several Java interfaces for various types of custom class loaders, which allow the class loaders to find and store classes in the shared class cache. These classes are in the com.ibm.oti.shared package.

The API documentation for this package is available here: API documentation

See “Using the Java Helper API” on page 358 for more information about how to use these interfaces.

Performance problems

Finding the root cause of a performance problem can be difficult, because many factors must be considered.

To learn more about debugging performance problems, see “Debugging performance problems” on page 191

Chapter 8. Security

The security components and utilities listed here are shipped with the IBM SDK for Java. The security components contain the IBM implementation of various security algorithms and mechanisms.

The following list summarizes the IBM security components and utilities that are available with the SDK. Further information about IBM security, including samples and API documentation, can be found here: IBM Security Information for Java.

- Java Certification path
- Java Authentication and Authorization Service (JAAS)
- Java Cryptographic Extension (JCE)
- Java Cryptographic Extension (JCE) FIPS
- IBM SecureRandom provider
- Java Generic Security Services (JGSS)
- Java Secure Socket Extension 2 (JSSE2)
- Public Key Cryptographic Standard (PKCS #11) Implementation Provider
- Simple Authentication and Security Layer (SASL)
- IBM Key Certificate Management
- Java XML Digital Signature
- Java XML Encryption
- IBM Common Access Card (CAC) provider
- iKeyman
- Keytool

Chapter 9. Troubleshooting and support

Use the information in this section to help you diagnose problems, run diagnostic tools, or submit a problem report.

Submitting problem reports

If you find a problem with Java, make a report through the product that supplied the Java SDK, or through the Operating System if there is no bundling product.

On z/OS, the 31-bit and 64-bit Java SDKs are bundled with WebSphere Application Server. These SDKs are also delivered as stand-alone z/OS program products:

- IBM 31-bit SDK for z/OS, Java Technology Edition, V7 (5655-W43)
- IBM 64-bit SDK for z/OS, Java Technology Edition, V7 (5655-W44.)

If you are using these standalone products, support is available through the normal z/OS operating system support structure. For more information about z/OS Java SDKs, see http://www-03.ibm.com/systems/z/os/zos/tools/java/.

There are several things you can try before submitting a Java problem to IBM. A useful starting point is the How Do I ...? page. In particular, the information about Troubleshooting problems might help you find and resolve the specific problem. If that does not work, try Looking for known problems.

If these steps have not helped you fix the problem, and you have an IBM support contract, consider Reporting the problem to IBM support. More information about support contracts for IBM products can be found in the Software Support Handbook.

If you do not have an IBM support contract, you might get informal support through other methods, described on the How Do I ...? page.

Problem determination

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

When you know what kind of problem you have, you might do one or more of the following tasks:

- Fix the problem
- Find a good workaround
- Collect the necessary data with which to generate a bug report to IBM

If your application runs on more than one platform and is exhibiting the same problem on them all, read the section about the platform to which you have the easiest access.

The chapters in this part are:

- "NLS problem determination" on page 194

First steps in problem determination

Before proceeding in problem determination, there are some initial questions to be answered.

Have you changed anything recently?
If you have changed, added, or removed software or hardware just before the problem occurred, back out the change and see if the problem persists.

What else is running on the workstation?
If you have other software, including a firewall, try switching it off to see if the problem persists.

Is the problem reproducible on the same workstation?
Knowing that this defect occurs every time the described steps are taken is helpful because it indicates a straightforward programming error. If the problem occurs at alternate times, or occasionally, thread interaction and timing problems in general are much more likely.

Is the problem reproducible on another workstation?
A problem that is not evident on another workstation might help you find the cause. A difference in hardware might make the problem disappear; for example, the number of processors. Also, differences in the operating system and application software installed might make a difference to the JVM. For example, the visibility of a race condition in the JVM or a user Java application might be influenced by the speed at which certain operations are performed by the system.

Does the problem occur on multiple platforms?
If the problem occurs only on one platform, it might be related to a platform-specific part of the JVM. Alternatively, it might be related to local code used inside a user application. If the problem occurs on multiple platforms, the problem might be related to the user Java application. Alternatively, it might be related to a cross-platform part of the JVM such as the Java Swing API. Some problems might be evident only on particular hardware; for example, Intel 32 bit architecture. A problem on particular hardware might indicate a JIT problem.

Can you reproduce the problem with the latest Service Refresh?
The problem might also have been fixed in a recent service refresh. Make sure that you are using the latest service refresh for your environment. Check the latest details on http://www.ibm.com/developerWorks/java/jdk.

Are you using a supported Operating System (OS) with the latest patches installed?
It is important to use an OS or distribution that supports the JVM and to have the latest patches for operating system components. For example, upgrading system libraries can solve problems. Moreover, later versions of system software can provide a richer set of diagnostic information. See *Setting up and checking environment* topics in the "Problem determination" on page 173 section, and check for latest details on the Developer Works Web site http://www.ibm.com/developerWorks.

Does turning off the JIT or AOT help?
If turning off the JIT or AOT prevents the problem, there might be a problem with the JIT or AOT. The problem can also indicate a race condition in your Java application that surfaces only in certain conditions. If the problem is

intermittent, reducing the JIT compilation threshold to 0 might help reproduce the problem more consistently. (See "JIT and AOT problem determination" on page 322.)

Have you tried reinstalling the JVM or other software and rebuilding relevant application files?
Some problems occur from a damaged or incorrect installation of the JVM or other software. It is also possible that an application might have inconsistent versions of binary files or packages. Inconsistency is likely in a development or testing environment and could potentially be solved by getting a fresh build or installation.

Is the problem particular to a multiprocessor (or SMP) platform? If you are working on a multiprocessor platform, does the problem still exist on a uniprocessor platform?
This information is valuable to IBM Service.

Have you installed the latest patches for other software that interacts with the JVM? For example, the IBM WebSphere Application Server and DB2®.
The problem might be related to configuration of the JVM in a larger environment, and might have been solved already in a fix pack. Is the problem reproducible when the latest patches have been installed?

Have you enabled core dumps?
Core dumps are essential to enable IBM Service to debug a problem. Core dumps are enabled by default for the Java process. See "Using dump agents" on page 221 for details. The operating system settings might also need to be in place to enable the dump to be generated and to ensure that it is complete. Details of the required operating system settings are contained in the relevant problem determination section for the platform.

What logging information is available?
The JVM logs information about problems as they occur. You can enable more detailed logging, and control where the logging information goes. For more details, see "JVM messages" on page 465.

z/OS problem determination

This section describes problem determination on z/OS.

The topics are:

Setting up and checking your z/OS environment

Set up the correct environment for the z/OS JVM to run correctly.

Maintenance:

The Java for z/OS Web site has up-to-date information about any changing operating system prerequisites for correct JVM operation. In addition, any new prerequisites are described in PTF HOLDDATA.

The Web site is at:

http://www.ibm.com/systems/z/os/zos/tools/java/

LE settings:

Language Environment (LE) Runtime Options (RTOs) affect operation of C and C++ programs such as the JVM. In general, the options provided by IBM using **C #pragma** statements in the code must not be overridden because they are generated as a result of testing to provide the best operation of the JVM.

Environment variables:

Environment variables that change the operation of the JVM in one release can be deprecated or change meaning in a following release. Therefore, you should review environment variables that are set for one release, to ensure that they still apply after any upgrade.

For information on compatibility between releases, see the Java on z/OS Web site at http://www.ibm.com/systems/z/os/zos/tools/java/.

Private storage usage:

The single most common class of failures after a successful installation of the SDK are those related to insufficient private storage.

As discussed in detail in "Understanding Memory Usage" on page 187, LE provides storage from Subpool 2, key 8 for C/C++ programs like the JVM that use C runtime library calls like malloc() to obtain memory. The LE HEAP refers to the areas obtained for all C/C++ programs that run in a process address space and request storage.

This area is used for the allocation of the Java heap where instances of Java objects are allocated and managed by Garbage Collection. The area is used also for any underlying allocations that the JVM makes during operations. For example, the JIT compiler obtains work areas for compilation of methods and to store compiled code.

Because the JVM must preallocate the maximum Java heap size so that it is contiguous, the total private area requirement is that of the maximum Java heap size that is set by the **-Xmx** option (or the 64 MB default if this is not set), plus an allowance for underlying allocations. A total private area of 140 MB is therefore a reasonable requirement for an instance of a JVM that has the default maximum heap size.

If the private area is restricted by either a system parameter or user exit, failures to obtain private storage occur. These failures show as OutOfMemoryErrors or Exceptions, failures to load libraries, or failures to complete subcomponent initialization during startup.

Setting up dumps:

The JVM generates a Javadump and System Transaction Dump (SYSTDUMP) when particular events occur.

The JVM, by default, generates the dumps when any of the following conditions occur:

- A SIGQUIT signal is received
- The JVM exits because of an error
- An unexpected native exception occurs (for example, a SIGSEGV, SIGILL, or SIGFPE signal is received)

You can use the **-Xdump** option to change the dumps that are produced on the various types of signal and the naming conventions for the dumps. For further details, see "Using dump agents" on page 221.

Failing transaction dumps (IEATDUMPs)

If a requested IEATDUMP cannot be produced, the JVM sends a message to the operator console. For example:

```
JVMDMP025I IEATDUMP failed RC=0x00000008 RSN=0x00000022 DSN=ABC.JVM.TDUMP.FUNGE2.
D070301.T171813
```

These return codes are fully documented in *z/OS V1R7.0 MVS Authorized Assembler Services Reference, 36.1.10 Return and Reason Codes*. Some common return codes are:

RC=0x00000008 RSN=0x00000022
Dump file name too long.

RC=0x00000008 RSN=0x00000026
Insufficient space for IEATDUMP.

RC=0x00000004
Partial dump taken. Typically, 2 GB size limit reached.

If the IEATDUMP produced is partial because of the 2 GB IEATDUMP size limit, use this message to trigger an SVC dump. To trigger the SVC dump, use a SLIP trap. For example:

```
SLIP SET,A=SVCD,J=FUNGE*,MSGID=JVMDMP025I,ID=JAVA,SDATA=(ALLPSA,NUC,SQA,RGN,LPA,
TRT,SUMDUMP),END
```

Multiple transaction dump (IEATDUMP) files on z/OS version 1.10 or newer

For z/OS version 1.10 or newer, on a 64-bit platform, IEATDUMP files are split into several smaller files if the IEATDUMP exceeds the 2 GB file size limit. Each file is given a sequence number.

If you specify a template for the IEATDUMP file name, append the `&DS` token to enable multiple dumps. The `&DS` token is replaced by an ordered sequence number, and must be at the end of the file name. For example, `X&DS` generates file names in the form `X001`, `X002`, and `X003`.

If you specify a template without the `&DS` token, `.X&DS` is appended automatically to the end of your template. If your template is too long to append `.X&DS`, a message is issued. The message advises that the template pattern is too long and that a default pattern will be used.

If you do not specify a template, the default template is used. The default template is:

```
%uid.JVM.%job.D%y%m%d.T%H%M%S.X&DS
```

You must merge the sequence of IEATDUMP files before IPCS can process the data. To merge the sequence of IEATDUMP files, use the TSO panel **IPCS** > **Utility** > **Copy MVS dump dataset**, or the **IPCS COPYDUMP** command. If you have copied or moved the IEATDUMP files from MVS to the z/OS UNIX System Services file system, you can use the "**cat**" command to merge the files, for example:

```
cat JVM.TDUMP.X001 JVM.TDUMP.X002 > JVM.TDUMP.FULL
```

For more information, see APAR: OA24232.

Note: For versions of z/OS before version 1.10, IEATDUMP file handling is unchanged.

General debugging techniques

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

In addition to the information given in this section, you can obtain z/OS publications from the IBM Web site. Go to http://www.ibm.com/support/publications/us/library/, and then choose the documentation link for your platform.

There are several diagnostic tools available with the JVM to help diagnose problems:

- Starting Javadumps, see "Using Javadump" on page 240.
- Starting Heapdumps, see "Using Heapdump" on page 262.
- Starting system dumps, see "Using system dumps and the dump viewer" on page 271.

z/OS provides various commands and tools that can be useful in diagnosing problems.

Using IPCS commands:

The Interactive Problem Control System (IPCS) is a tool provided in z/OS to help you diagnose software failures. IPCS provides formatting and analysis support for dumps and traces produced by z/OS.

Here are some sample IPCS commands that you might find useful during your debugging sessions. In this case, the address space of interest is `ASID(x'7D')`.

ip verbx ledata 'nthreads(*)'
: This command provides the stack traces for the TCBs in the dump.

ip setd asid(x'007d')
: This command is to set the default ASID; for example, to set the default asid to x'007d'.

ip verbx ledata 'all,asid(007d),tcb(tttttt)'
: In this command, the **all** report formats out key LE control blocks such as CAA, PCB, ZMCH, CIB. In particular, the CIB/ZMCH captures the PSW and GPRs at the time the program check occurred.

ip verbx ledata 'cee,asid(007d),tcb(tttttt)'
: This command formats out the traceback for one specific thread.

ip summ regs asid(x'007d')
: This command formats out the TCB/RB structure for the address space. It is rarely useful for JVM debugging.

ip verbx sumdump
: Then issue `find 'slip regs sa'` to locate the GPRs and PSW at the time a SLIP TRAP is matched. This command is useful for the case where you set a SA (Storage Alter) trap to catch an overlay of storage.

ip omvsdata process detail asid(x'007d')
: This command generates a report for the process showing the thread status from a USS kernel perspective.

ip select all
: This command generates a list of the address spaces in the system at the time of the dump, so that you can tie up the ASID with the JOBNAME.

ip systrace asid(x'007d') time(gmt)
: This command formats out the system trace entries for all threads in this address space. It is useful for diagnosing loops. `time(gmt)` converts the TOD Clock entries in the system trace to a human readable form.

For further information about IPCS, see the z/OS documentation (*z/OS V1R7.0 MVS IPCS Commands*).

Using dbx:

The dbx utility has been improved for z/OS V1R6. You can use dbx to analyze transaction (or system) dumps and to debug a running application.

For information about dbx, see the z/OS documentation; *z/OS V1R6.0 UNIX System Services Programming Tools* at http://publibz.boulder.ibm.com/epubs/pdf/bpxza630.pdf.

Interpreting error message IDs:

While working in the OMVS, if you get an error message and want to understand exactly what the error message means there is a Web site you can go to.

Go to: http://www-03.ibm.com/systems/z/os/zos/bkserv/lookat/index.html and enter the message ID. Then select your OS level and then press enter. The output will give a better understanding of the error message. To decode the `errno2` values, use the following command:

```
bpxmtext <reason_code>
```

`Reason_code` is specified as 8 hexadecimal characters. Leading zeros can be omitted.

Diagnosing crashes

A crash should occur only because of a fault in the JVM, or because of a fault in native (JNI) code that is being run inside the Java process. A crash is more strictly defined on z/OS as a program check that is handled by z/OS UNIX as an unrecoverable signal (for example, SIGSEGV for PIC4; 10, 11, or SIGILL for PIC1).

Documents to gather:

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

When one of these unrecoverable signals occurs, the JVM Signal Handler takes control. The default action of the signal handler is to produce a transaction dump (through the BCP IEATDUMP service), a JVM snap trace dump, and a formatted Javadump. Output should be written to the message stream that is written to

stderr in the form of:

```
Unhandled exception
Type=Segmentation error vmState=0x00000000
Target=2_30_20060227_05498_bHdSMr (z/OS 01.06.00)
CPU=s390 (2 logical CPUs) (0x180000000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=00000000 Signal_Code=00000035
Handler1=115F8590 Handler2=116AFC60
gpr0=00000064 gpr1=00000000 gpr2=117A3D70 gpr3=00000000
gpr4=114F5280 gpr5=117C0E28 gpr6=117A2A18 gpr7=9167B460
gpr8=0000007E gpr9=116AF5E8 gpr10=1146E21C gpr11=0000007E
gpr12=1102C7D0 gpr13=11520838 gpr14=115F8590 gpr15=00000000
psw0=078D0400 psw1=917A2A2A
fpr0=48441040 fpr1=3FFF1999 fpr2=4E800001 fpr3=99999999
fpr4=45F42400 fpr5=3FF00000 fpr6=00000000 fpr7=00000000
fpr8=00000000 fpr9=00000000 fpr10=00000000 fpr11=00000000
fpr12=00000000 fpr13=00000000 fpr14=00000000 fpr15=00000000
Program_Unit_Name=
Program_Unit_Address=1167B198 Entry_Name=j9sig_protect
Entry_Address=1167B198
JVMDUMP006I Processing Dump Event "gpf", detail "" - Please Wait.
JVMDUMP007I JVM Requesting System Dump using 'CHAMBER.JVM.TDUMP.CHAMBER1.D060309.T144842'
IEATDUMP in progress with options SDATA=(LPA,GRSQ,LSQA,NUC,PSA,RGN,SQA,SUM,SWA,TRT)
IEATDUMP success for DSN='CHAMBER.JVM.TDUMP.CHAMBER1.D060309.T144842'
JVMDUMP010I System Dump written to CHAMBER.JVM.TDUMP.CHAMBER1.D060309.T144842
JVMDUMP007I JVM Requesting Snap Dump using '/u/chamber/test/ras/Snap0001.20060309.144842.196780.trc'
JVMDUMP010I Snap Dump written to /u/chamber/test/ras/Snap0002.20060309.144842.196780.trc
JVMDUMP007I JVM Requesting Java Dump using '/u/chamber/test/ras/javacore.20060309.144842.196780.txt'
JVMDUMP010I Java Dump written to /u/chamber/test/ras/javacore.20060309.144842.196780.txt
JVMDUMP013I Processed Dump Event "gpf", detail "".
```

The output shows the location in HFS into which the Javadump file was written and the name of the MVS data set to which the transaction dump is written. These locations are configurable and are described in "Overview of the available diagnostic tools" on page 212 and "Using dump agents" on page 221.

These documents provide the ability to determine the failing function, and therefore decide which product owns the failing code, be it the JVM, application JNI code, or native libraries acquired from another vendor (for example native JDBC drivers).

The JVM will display error messages if it is unable to produce the dumps. The IEATDUMP error return codes, `RC=...` and `RSN=...`, are included in the messages. These return codes are fully documented in *z/OS V1R7.0 MVS Authorized Assembler Services Reference, 36.1.10 Return and Reason Codes*.

This example shows the error messages displayed when there is insufficient disk space to write the IEATDUMP:

```
JVMDUMP007I JVM Requesting System dump using 'J9BUILD.JVM.TDUMP.SSHD1.D080326.T081447'
IEATDUMP in progress with options SDATA=(LPA,GRSQ,LSQA,NUC,PSA,RGN,SQA,SUM,SWA,TRT)
IEATDUMP failure for DSN='J9BUILD.JVM.TDUMP.SSHD1.D080326.T081447' RC=0x00000008 RSN=0x00000026
JVMDUMP012E Error in System dump: J9BUILD.JVM.TDUMP.SSHD1.D080326.T081447
```

When an IEATDUMP fails, an error message is also written to the operator console. If the IEATDUMP fails because of the 2 GB IEATDUMP size limit, you can use a SLIP trap to trigger an SVC DUMP to ensure all the required diagnostics information is available. See "Setting up dumps" on page 176 for more information.

z/OS V1R7.0 MVS Authorized Assembler Services Reference is available at http://www-03.ibm.com/systems/z/os/zos/bkserv/r12pdf/#mvs.

Determining the failing function:

The most practical way to find where the exception occurred is to review either the CEEDUMP or the Javadump. Both of these reports show where the exception occurred and the native stack trace for the failing thread.

The same information can be obtained from the transaction dump by using either the dump viewer (see "Using system dumps and the dump viewer" on page 271), the dbx debugger, or the IPCS LEDATA VERB Exit.

The CEEDUMP shows the C-Stack (or native stack, which is separate from the Java stack that is built by the JVM). The C-stack frames are also known on z/OS as Dynamic Storage Areas (DSAs), because a DSA is the name of the control block that LE provides as a native stack frame for a C/C++ program. The following traceback from a CEEDUMP shows where a failure occurred:

```
Traceback:
    DSA      Entry       E  Offset  Load Mod  Program Unit  Service  Status
    00000001 __cdump     +00000000  CELQLIB                HLE7709  Call
    00000002 @@WRAP@MULTHD
                         +00000266  CELQLIB                         Call
    00000003 j9dump_create
                         +0000035C  *PATHNAM               j040813  Call
    00000004 doSystemDump+0000008C  *PATHNAM               j040813  Call
    00000005 triggerDumpAgents
                         +00000270  *PATHNAM               j040813  Call
    00000006 vmGPHandler +00000C4C  *PATHNAM               j040813  Call
    00000007 gpHandler   +000000D4  *PATHNAM               j040813  Call
    00000008 __zerro     +00000BC4  CELQLIB                HLE7709  Call
    00000009 __zerros    +0000016E  CELQLIB                HLE7709  Call
    0000000A CEEHDSP     +00003A2C  CELQLIB   CEEHDSP      HLE7709  Call
    0000000B CEEOSIGJ    +00000956  CELQLIB   CEEOSIGJ     HLE7709  Call
    0000000C CELQHROD    +00000256  CELQLIB   CELQHROD     HLE7709  Call
    0000000D CEEOSIGG    -08B3FBBC  CELQLIB   CEEOSIGG     HLE7709  Call
    0000000E CELQHROD    +00000256  CELQLIB   CELQHROD     HLE7709  Call
    0000000F Java_dumpTest_runTest
                         +00000044  *PATHNAM                        Exception
    00000010 RUNCALLINMETHOD
                         -0000F004  *PATHNAM                        Call
    00000011 gpProtectedRunCallInMethod
                         +00000044  *PATHNAM               j040813  Call
    00000012 j9gp_protect+00000028  *PATHNAM               j040813  Call
    00000013 gpCheckCallin
                         +00000076  *PATHNAM               j040813  Call
    00000014 callStaticVoidMethod
                         +00000098  *PATHNAM               j040813  Call
    00000015 main        +000029B2  *PATHNAM               j904081  Call
    00000016 CELQINIT    +00001146  CELQLIB   CELQINIT     HLE7709  Call

    DSA      DSA Addr          E  Addr             PU Addr           PU Offset   Comp Date Attributes
    00000001 00000001082F78E0  000000001110EB38    0000000000000000  ********    20040312  XPLINK  EBCDIC  POSIX  IEEE
    00000002 00000001082F7A20  00000000110AF458    0000000000000000  ********    20040312  XPLINK  EBCDIC  POSIX  Floating Point
    00000003 00000001082F7C00  0000000011202988    0000000000000000  ********    20040817  XPLINK  EBCDIC  POSIX  IEEE
    00000004 00000001082F8100  0000000011213770    0000000000000000  ********    20040817  XPLINK  EBCDIC  POSIX  IEEE
    00000005 00000001082F8200  0000000011219760    0000000000000000  ********    20040817  XPLINK  EBCDIC  POSIX  IEEE
    00000006 00000001082F8540  000000007CD4BDA8    0000000000000000  ********    20040817  XPLINK  EBCDIC  POSIX  IEEE
    00000007 00000001082F9380  00000000111FF190    0000000000000000  ********    20040817  XPLINK  EBCDIC  POSIX  IEEE
    00000008 00000001082F9480  00000000111121E0    0000000000000000  ********    20040312  XPLINK  EBCDIC  POSIX  IEEE
    00000009 00000001082FA0C0  0000000011112048    0000000000000000  ********    20040312  XPLINK  EBCDIC  POSIX  IEEE
    0000000A 00000001082FA1C0  0000000010DB8EA0    0000000010DB8EA0  00003A2C    20040312  XPLINK  EBCDIC  POSIX  Floating Point
    0000000B 00000001082FCAE0  0000000010E3D530    0000000010E3D530  00000956    20040312  XPLINK  EBCDIC  POSIX  Floating Point
    0000000C 00000001082FD4E0  0000000010D76778    0000000010D76778  00000256    20040312  XPLINK  EBCDIC  POSIX  Floating Point
    0000000D 00000001082FD720  0000000010E36C08    0000000010E36C08  08B3FBB0    20040312  XPLINK  EBCDIC  POSIX  Floating Point
    0000000E 00000001082FE540  0000000010D76778    0000000010D76778  00000256    20040312  XPLINK  EBCDIC  POSIX  Floating Point
    0000000F 00000001082FE780  00000000122C66B0    0000000000000000  ********    20040802  XPLINK  EBCDIC  POSIX  IEEE
    00000010 00000001082FE880  000000007CD28030    0000000000000000  ********    ^C"^22^04^FF^FDu^58  XPLINK  EBCDIC  POSIX  IEEE
    00000011 00000001082FEC80  000000007CD515B8    0000000000000000  ********    20040817  XPLINK  EBCDIC  POSIX  IEEE
    00000012 00000001082FED80  00000000111FF948    0000000000000000  ********    20040817  XPLINK  EBCDIC  POSIX  IEEE
    00000013 00000001082FEE80  000000007CD531A8    0000000000000000  ********    20040817  XPLINK  EBCDIC  POSIX  IEEE
    00000014 00000001082FEF80  000000007CD4F148    0000000000000000  ********    20040817  XPLINK  EBCDIC  POSIX  IEEE
```

Note:

1. The stack frame that has a status value of `Exception` indicates where the crash occurred. In this example, the crash occurs in the function `Java_dumpTest_runTest`.

2. The value under Service for each DSA is the service string. The string is built in the format of `jyymmdd`, where `j` is the identifier for the code owner and `yymmdd` is the build date. A service string with this format indicates that the function is part of the JVM. All functions have the same build date, unless you have been supplied with a dll by IBM Service for diagnostic or temporary fix purposes.

Working with TDUMPs using IPCS:

A TDUMP or Transaction Dump is generated from the MVS service IEATDUMP by default in the event of a program check or exception in the JVM. You can disable the generation of a TDUMP, but it is not recommended by IBM Service.

A TDUMP can contain multiple Address Spaces. It is important to work with the correct address space associated with the failing java process.

To work with a TDUMP in IPCS, here is a sample set of steps to add the dump file to the IPCS inventory:

1. Browse the dump data set to check the format and to ensure that the dump is correct.
2. In IPCS option **3 (Utility Menu)**, suboption **4 (Process list of data set names)** type in the TSO HLQ (for example, `DUMPHLQ`) and press Enter to list data sets. You must `ADD` (`A` in the command-line alongside the relevant data set) the uncompressed (untersed) data set to the IPCS inventory.
3. You can select this dump as the default one to analyze in two ways:
   - In IPCS option **4 (Inventory Menu)** type `SD` to add the selected data set name to the default globals.
   - In IPCS option **0 (DEFAULTS Menu)**, change Scope and Source

     ```
     Scope ==> BOTH (LOCAL, GLOBAL, or BOTH)

     Source ==> DSNAME('DUMPHLQ.UNTERSED.SIGSEGV.DUMP')
     Address Space ==>
     Message Routing ==> NOPRINT TERMINAL
     Message Control ==> CONFIRM VERIFY FLAG(WARNING)
     Display Content ==> NOMACHINE REMARK REQUEST NOSTORAGE SYMBOL
     ```

     If you change the Source default, IPCS displays the current default address space for the new source and ignores any data entered in the address space field.
4. To initialize the dump, select one of the analysis functions, such as IPCS option **2.4 SUMMARY - Address spaces and tasks**, which will display something like the following output and give the TCB address. (Note that non-zero CMP entries reflect the termination code.)

   ```
   TCB: 009EC1B0
      CMP..... 940C4000  PKF..... 80        LMP..... FF        DSP..... 8C
      TSFLG.... 20        STAB..... 009FD420  NDSP..... 00002000
      JSCB..... 009ECCB4  BITS..... 00000000  DAR..... 00
      RTWA..... 7F8BEDF0  FBYT1.... 08
      Task non-dispatchability flags from TCBFLGS5:
       Secondary non-dispatchability indicator
      Task non-dispatchability flags from TCBNDSP2:
       SVC Dump is executing for another task

   SVRB: 009FD9A8
      WLIC..... 00000000  OPSW..... 070C0000  81035E40
      LINK..... 009D1138

   PRB: 009D1138
      WLIC..... 00040011  OPSW..... 078D1400  B258B108
   ```

```
   LINK..... 009ECBF8
   EP....... DFSPCJB0  ENTPT.... 80008EF0

PRB: 009ECBF8
   WLIC..... 00020006  OPSW..... 078D1000  800091D6
   LINK..... 009ECC80
```

Useful IPCS commands and some sample output:

Some IPCS commands that you can use when diagnosing crashes.

In IPCS option **6 (COMMAND Menu)** type in a command and press the Enter key:

ip st
Provides a status report.

ip select all
Shows the Jobname to ASID mapping:

```
ASID JOBNAME  ASCBADDR  SELECTION CRITERIA
  ---- -------- --------  ------------------
  0090 H121790  00EFAB80  ALL
  0092 BPXAS    00F2E280  ALL
  0093 BWASP01  00F2E400  ALL
  0094 BWASP03  00F00900  ALL
  0095 BWEBP18  00F2EB80  ALL
  0096 BPXAS    00F8A880  ALL
```

ip systrace all time(local)
Shows the system trace:

```
PR ASID,WU-ADDR-  IDENT CD/D PSW----- ADDRESS-  UNIQUE-1 UNIQUE-2 UNIQUE-3
                                                UNIQUE-4 UNIQUE-5 UNIQUE-6

  09-0094 009DFE88   SVCR    6 078D3400 8DBF7A4E  8AA6C648 0000007A 24AC2408
  09-0094 05C04E50   SRB       070C0000 8AA709B8  00000094 02CC90C0 02CC90EC
                                                  009DFE88 A0
  09-0094 05C04E50   PC    ...   0      0AA70A06           0030B
  09-0094 00000000   SSRV  132          00000000  0000E602 00002000 7EF16000
                                                  00940000
```

For suspected loops you might need to concentrate on ASID and exclude any branch tracing:

```
ip systrace asid(x'3c') exclude(br)
```

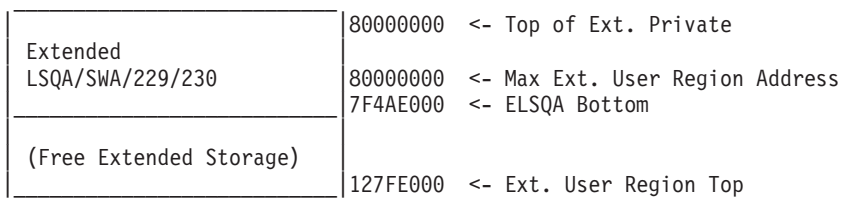
ip summ format asid(x'94')
To find the list of TCBs, issue a find command for "T C B".

ip verbx ledata 'ceedump asid(94) tcb(009DFE88)'
Obtains a traceback for the specified TCB.

ip omvsdata process detail asid(x'94')
Shows a USS perspective for each thread.

ip verbx vsmdata 'summary noglobal'
Provides a memory usage report:

```
LOCAL STORAGE MAP
```



```
        |                              |
        |  Extended User Region        |
        |______________________________|10D00000  <- Ext. User Region Start
        :                              :
        : Extended Global Storage      :
        ======================================<- 16M Line
        : Global Storage               :
        :______________________________:  A00000  <- Top of Private
        |                              |
        |  LSQA/SWA/229/230            |  A00000  <- Max User Region Address
        |______________________________|  9B8000  <- LSQA Bottom
        |                              |
        |  (Free Storage)              |
        |______________________________|    7000  <- User Region Top
        |                              |
        |  User Region                 |
        |______________________________|    6000  <- User Region Start
        : System Storage               :
        :______________________________:       0
```

ip verbx ledata 'nthreads(*)'
Obtains the tracebacks for all threads.

ip status regs
Shows the PSW and registers:

```
CPU STATUS:
 BLS18058I Warnings regarding STRUCTURE(Psa) at ASID(X'0001') 00:
 BLS18300I    Storage not in dump
 PSW=00000000 00000000
     (Running in PRIMARY  key 0  AMODE 24  DAT OFF)
     DISABLED FOR PER I/O EXT MCH
   ASCB99 at FA3200  JOB(JAVADV1)  for the home ASID
   ASXB99 at 8FDD00 and TCB99G at 8C90F8 for the home ASID
   HOME ASID: 0063 PRIMARY ASID: 0063 SECONDARY ASID: 0063

   General purpose register values
     Left halves of all registers contain zeros
      0-3  00000000  00000000  00000000  00000000
      4-7  00000000  00000000  00000000  00000000
      8-11 00000000  00000000  00000000  00000000
     12-15 00000000  00000000  00000000  00000000

   Access register values
       0-3  00000000  00000000  00000000  00000000
       4-7  00000000  00000000  00000000  00000000
       8-11 00000000  00000000  00000000  00000000
      12-15 00000000  00000000  00000000  00000000

   Control register values
      0-1  00000000_5F04EE50  00000001_FFC3C007
      2-3  00000000_5A057800  00000001_00C00063
      4-5  00000000_00000063  00000000_048158C0
      6-7  00000000_00000000  00000001_FFC3C007
      8-9  00000000_00000000  00000000_00000000
     10-11 00000000_00000000  00000000_00000000
     12-13 00000000_0381829F  00000001_FFC3C007
     14-15 00000000_DF884811  00000000_7F5DC138
```

ip cbf rtct
Helps you to find the ASID by looking at the ASTB mapping to see which ASIDs are captured in the dump.

ip verbx vsmdata 'nog summ'
Provides a summary of the virtual storage management data areas:

```
DATA FOR SUBPOOL   2  KEY  8 FOLLOWS:

 -- DQE LISTING (VIRTUAL BELOW, REAL ANY64)

  DQE:  ADDR 12C1D000 SIZE    32000
  DQE:  ADDR 1305D000 SIZE   800000
  DQE:  ADDR 14270000 SIZE   200000
  DQE:  ADDR 14470000 SIZE 10002000
  DQE:  ADDR 24472000 SIZE   403000
  DQE:  ADDR 24875000 SIZE   403000
  DQE:  ADDR 24C78000 SIZE    83000
  DQE:  ADDR 24CFB000 SIZE   200000
  DQE:  ADDR 250FD000 SIZE   39B000
                                    FQE: ADDR 25497028 SIZE      FD8
  DQE:  ADDR 25498000 SIZE   735000
                                    FQE: ADDR 25BCC028 SIZE      FD8
  DQE:  ADDR 25D36000 SIZE   200000
  DQE:  ADDR 29897000 SIZE   200000
  DQE:  ADDR 2A7F4000 SIZE   200000
  DQE:  ADDR 2A9F4000 SIZE   200000
  DQE:  ADDR 2AC2F000 SIZE   735000
                                    FQE: ADDR 2B363028 SIZE      FD8
  DQE:  ADDR 2B383000 SIZE   200000
  DQE:  ADDR 2B5C7000 SIZE   200000
  DQE:  ADDR 2B857000 SIZE     1000


***** SUBPOOL   2  KEY  8 TOTAL ALLOC: 132C3000 ( 00000000 BELOW, 132C3000
```

ip verbx ledata 'all asid(54) tcb(009FD098)'
Finds the PSW and registers at time of the exception:

```
+000348 MCH_EYE:ZMCH
+000350 MCH_GPR00:00000000 000003E7   MCH_GPR01:00000000 00000000
+000360 MCH_GPR02:00000001 00006160   MCH_GPR03:00000000 00000010
+000370 MCH_GPR04:00000001 082FE780   MCH_GPR05:00000000 000000C0
+000380 MCH_GPR06:00000000 00000000   MCH_GPR07:00000000 127FC6E8
+000390 MCH_GPR08:00000000 00000007   MCH_GPR09:00000000 127FC708
+0003A0 MCH_GPR10:00000001 08377D70   MCH_GPR11:00000001 0C83FB78
+0003B0 MCH_GPR12:00000001 08300C60   MCH_GPR13:00000001 08377D00
+0003C0 MCH_GPR14:00000000 112100D0   MCH_GPR15:00000000 00000000
+0003D0 MCH_PSW:07852401 80000000 00000000 127FC6F8     MCH_ILC:0004
+0003E2 MCH_IC1:00  MCH_IC2:04  MCH_PFT:00000000 00000000
+0003F0 MCH_FLT_0:48410E4F 6C000000 4E800001 31F20A8D
+000400 MCH_FLT_2:406F0000 00000000 00000000 00000000
+000410 MCH_FLT_4:45800000 00000000 3FF00000 00000000
+000420 MCH_FLT_6:00000000 00000000 00000000 00000000
+0004B8 MCH_EXT:00000000 00000000
```

blscddir dsname('DUMPHLQ.ddir')
Creates an IPCS DDIR.

runc addr(2657c9b8) link(20:23) chain(9999) le(x'1c') *or* **runc addr(25429108) structure(tcb)**
Runs a chain of control blocks using the RUNCHAIN command.

`addr`: the start address of the first block

`link`: the link pointer start and end bytes in the block (decimal)

`chain`: the maximum number of blocks to be searched (default=999)

`le`: the length of data from the start of each block to be displayed (hex)

`structure`: control block type

Debugging hangs

A hang refers to a process that is still present, but has become unresponsive.

This lack of response can be caused by any one of these reasons:

- The process has become deadlocked, so no work is being done. Usually, the process is taking up no CPU time.
- The process has become caught in an infinite loop. Usually, the process is taking up high CPU time.
- The process is running, but is suffering from very bad performance. This is not an actual hang, but is often initially mistaken for one.

The process is deadlocked:

A deadlocked process does not use any CPU time.

You can monitor this condition by using the USS **ps** command against the Java process:

```
    UID        PID       PPID  C    STIME TTY       TIME CMD
CBAILEY        253        743  - 10:24:19 ttyp0003  2:34 java -classpath .Test2Frame
```

If the value of `TIME` increases in a few minutes, the process is still using CPU, and is not deadlocked.

For an explanation of deadlocks and how the Javadump tool is used to diagnose them, see "Locks, monitors, and deadlocks (LOCKS)" on page 250.

The process is looping:

If no deadlock exists between threads and the process appears to be hanging but is consuming CPU time, look at the work the threads are doing. To do this, take a console-initiated dump (SVC dump).

Follow these steps to take a console-initiated dump:

1. Use the operating system commands **(D OMVS,A=ALL)** or **SDSF (DA = Display Active)** to locate the ASID of interest.
2. Use the **DUMP** command to take a console-initiated dump both for hangs and for loops:

   ```
   DUMP COMM=(Dump for problem 12345)
   R xx,ASID=(53,d),DSPNAME=('OMVS '.*),CONT
   R yy,SDATA=(GRSQ,LSQA,RGN,SUM,SWA,TRT,LPA,NUC,SQA)
   ```

   Prefix all commands on the SDSF panels with a forward slash (/). The console responds to the **DUMP** command with a message requesting additional operands, and provides you with a 2-digit reply ID. You supply the additional operands using the **R** (reply) command, specifying the reply ID (shown as `xx` or `yy` in the previous example). You can use multiple replies for the operands by specifying the CONT operand, as in the previous example.

You can select the process to dump using the z/OS job name instead of the ASID:

```
R xx,JOBNAME=SSHD9,CONT
```

When the console dump has been generated, you can view the Systrace in IPCS to identify threads that are looping. You can do this in IPCS as follows:

```
ip systrace asid(x'007d') time(gmt)
```

This command formats out the system trace entries for all threads that are in address space `0x7d`. The `time(gmt)` option converts the TOD clock entries, which are in the system trace, to a human readable form.

From the output produced, you can determine which are the looping threads by identifying patterns of repeated CLCK and EXT1005 interrupt trace entries, and subsequent redispatch DSP entries. You can identify the instruction address range of the loop from the PSWs (Program Status Words) that are traced in these entries.

You can also analyze z/OS console (SVC) dumps using the system dump viewer provided in the SDK, see "Using system dumps and the dump viewer" on page 271.

The process is performing badly:

If you have no evidence of a deadlock or an infinite loop, the process is probably suffering from very bad performance. Bad performance can be caused because threads have been placed into explicit sleep calls, or by excessive lock contention, long garbage collection cycles, or for several other reasons. This condition is not a hang and should be handled as a performance problem.

See "Debugging performance problems" on page 191 for more information.

Understanding Memory Usage

To debug memory leaks you need to understand the mechanisms that can cause memory problems, how the JVM uses the LE HEAP, how the JVM uses z/OS virtual storage, and the possible causes of a java.lang.OutOfMemoryError exception.

Memory problems can occur in the Java process through two mechanisms:

- A native (C/C++) memory leak that causes increased usage of the LE HEAP, which can be seen as excessive usage of Subpool2, Key 8, or storage, and an excessive Working Set Size of the process address space
- A Java object leak in the Java-managed heap. The leak is caused by programming errors in the application or the middleware. These object leaks cause an increase in the amount of live data that remains after a garbage collection cycle has been completed.

Allocations to LE HEAP:

The Java process makes two distinct allocation types to the LE HEAP.

The first type is the allocation of the Java heap that garbage collection manages. The Java heap is allocated during JVM startup as a contiguous area of memory. Its size is that of the maximum Java heap size parameter. Even if the minimum, initial, heap size is much smaller, you must allocate for the maximum heap size to ensure that one contiguous area will be available should heap expansion occur.

The second type of allocation to the LE HEAP is that of calls to malloc() by the JVM, or by any native JNI code that is running under that Java process. This includes application JNI code, and vendor-supplied native libraries; for example, JDBC drivers.

z/OS virtual storage:

To debug these problems, you must understand how C/C++ programs, such as the JVM, use virtual storage on z/OS. To do this, you need some background understanding of the z/OS Virtual Storage Management component and LE.

The process address space on 31-bit z/OS has 31-bit addressing that allows the addressing of 2 GB of virtual storage. The process address space on 64-bit z/OS has 64-bit addressing that allows the addressing of over 2 GB of virtual storage. This storage includes areas that are defined as common (addressable by code running in all address spaces) and other areas that are private (addressable by code running in that address space only).

The size of common areas is defined by several system parameters and the number of load modules that are loaded into these common areas. On many typical systems, the total private area available is about 1.4 GB. From this area, memory resources required by the JVM and its subcomponents such as the JIT are allocated by calls to malloc(). These resources include the Java heap and memory required by application JNI code and third-party native libraries.

A Java OutOfMemoryError exception typically occurs when the Java heap is exhausted. For further information on z/OS storage allocation, see: http://www.redbooks.ibm.com/redbooks/SG247035/. It is possible for a 31-bit JVM to deplete the private storage area, resulting in and OutOfMemoryError exception. For more information, see: "OutOfMemoryError exceptions."

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

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

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

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

OutOfMemoryError exceptions:

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 "Memory management" on page 23 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 334.

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 "Available tools for processing Heapdumps" on page 263.

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 http://publib.boulder.ibm.com/infocenter/zos/v1r10/topic/com.ibm.zos.r10.bpxb200/shbene.htm.

Tracing leaks:

Some useful techniques for tracing leaks are built into the JVM.

The techniques are:

- The **-verbose:gc** option.
- HPROF tools. See "Using the HPROF Profiler" on page 385.

–Xrunjnichk option:

You can use the **-Xrunjnichk** option to trace JNI calls that are made by your JNI code or by any JVM components that use JNI. This helps you to identify incorrect uses of JNI libraries from native code and can help you to diagnose JNI memory leaks.

JNI memory leaks occur when a JNI thread allocates objects and fails to free them. The Garbage Collector does not have enough information about the JNI thread to know when the object is no longer needed. For more information, see "The JNI and the Garbage Collector" on page 90.

Note that **-Xrunjnichk** is equivalent to **-Xcheck:jni**. See "Debugging the JNI" on page 97 for information on the **-Xrunjnichk** suboptions.

–Xcheck:memory option:

The **-Xcheck:memory** option can help you identify memory leaks inside the JVM. The **-Xcheck:memory** option traces the JVM calls to the operating system's malloc() and free() functions, and identifies any JVM mistakes in memory allocation.

The system property **-Dcom.ibm.dbgmalloc=true** provides memory allocation information about class library native code. Use this system property with the **-Xcheck:memory:callsite=1000** option to obtain detailed information about class library callsites and their allocation sizes. Here is some sample output:

```
| total alloc   | total freed   | delta alloc   | delta freed   | high water | largest
| blocks| bytes | blocks| bytes | blocks| bytes | blocks| bytes | blocks| bytes | bytes | num   | callsite
|-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+------------
|     125   16000       0       0       0       0       0       0     125   16000     128       1 dbgwrapper/dbgmalloc.c:434
|       1    3661       1    3661       0       0       0       0       1    3661    3661       1 java/TimeZone_md.c:294
|    4144 18121712    4144 18121712     420 1836660     420 1836660     2    8746    4373       1 java/UnixFileSystem_md.c:373
|      10     124      10     124       0       0       0       0       2      55      51       1 java/jni_util.c:874
|       2   80797       2   80797       0       0       0       0       1   64413   64413       2 java/io_util.c:102
|       1      52       1      52       0       0       0       0       1      52      52       1 jli/java.c:2472
|       2    1872       1     264       0       0       0       0       2    1872    1608       2 net/linux_close.c:135
|       9     288       9     288       0       0       0       0       2      64      32       1 net/Inet6AddressImpl.c:280
|   99524 3260992980 99524 3260992980 10514 344503782  10515 344536549  1   32767   32767       1 net/SocketInputStream.c:93
|       3      24       3      24       0       0       1       8       2      16       8       1 net/linux_close.c:276
|     201    4824       0       0       0       0       0       0     201    4824      24       1 net/linux_close.c:149
|     311 1003152     261  496080       0       0      68  142128     119  651040  261360      45 zip/zip_util.c:655
|     311   31100     261   26100       0       0      68    6800     119   11900     100       1 zip/zip_util.c:230
```

```
|      243   15552     222   14208       0       0      85    5440     160   10240      64       1 zip/Inflater.c:61
|        1      64       1      64       0       0       0       0       1      64      64       1 zip/Deflater.c:76
|      146    7592     123    6396       3     156      74    3848      97    5044      52       1 zip/zip_util_md.c:75
|     3242 1443289    3241 1439991      25    4000      92  252856      71  262264    8192     679 zip/zip_util.c:917
|      311  125140     261   61724       0       0      68   17856     119   81500   32668      45 zip/zip_util.c:657
|      146   37376     123   31488       3     768      74   18944      97   24832     256       1 zip/zip_util_md.c:132
```

For more information about setting **-Dcom.ibm.dbgmalloc=true**, see "System property command-line options" on page 419.

For more information about the **-Xcheck:memory** option, see "JVM command-line options" on page 428.

Using Heapdump to debug memory leaks:

You can use Heapdump to analyze the Java Heap.

For details about analyzing the Heap, see "Using Heapdump" on page 262.

Debugging performance problems

Locating the causes of poor performance is often difficult. Although many factors can affect performance, the overall effect is generally perceived as poor response or slow execution of your program.

Correcting one performance problem might cause more problems in another area. By finding and correcting a bottleneck in one place you might only shift the cause of poor performance to other areas. To improve performance, experiment with tuning different parameters, monitoring the effect, and retuning until you are satisfied that your system is performing acceptably

Finding the bottleneck:

The aspects of the system that you are most interested in measuring are CPU usage and memory usage. It is possible that even after extensive tuning efforts the CPU is not powerful enough to handle the workload, in which case a CPU upgrade is required. Similarly, if the program is running in an environment in which it does not have enough memory after tuning, you must increase memory size.

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

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

z/OS systems resource usage:

The z/OS Resource Measurement Facility (RMF™) gives a detailed view of z/OS processor, memory, and I/O device performance metrics.

JVM heap sizing:

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

The Java heap size is one of the most important tuning parameters of your JVM. See "How to do heap sizing" on page 45 for information on how to correctly set the size of your heap.

JIT compilation and performance:

The JIT is another area that can affect the performance of your program. When deciding whether or not to use JIT compilation, you must make a balance between faster execution and increased processor usage during compilation .

The performance of short-running applications can be improved by using the **-Xquickstart** command-line parameter. The JIT is switched on by default, but you can use **-Xint** to turn it off. You also have considerable flexibility in controlling JIT processing. For more details about the JIT, see "The JIT compiler" on page 57 and "JIT and AOT problem determination" on page 322.

IBM Monitoring and Diagnostic Tools for Java:

The IBM Monitoring and Diagnostic Tools for Java are a set of GUI-based tools for monitoring Java applications and analyzing diagnostic data. These tools are designed to make Java diagnostic tasks as quick and as easy as possible.

Some tools can be attached to a running JVM, to monitor application behavior and resource usage. For other tools, you generate dump files from your system or JVM, then analyze the file in the tool. By using the tools, you can diagnose problems such as memory leaks, thread contention issues, and I/O bottlenecks, as well as getting information and recommendations to help you tune the JVM and improve the performance of your application.

For more information about the tools, see "Using the IBM Monitoring and Diagnostic Tools for Java" on page 219.

Testing JVM optimizations:

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

Java monitor optimizations

This release introduces new optimizations that are expected to improve CPU efficiency. However, there might be some situations where reduced 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
  ```

- 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
```

Lock optimizations

This release introduces new locking optimizations that 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:

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

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

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

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

MustGather information for z/OS

The more information that you can collect about a problem, the easier it is to diagnose that problem. A large set of data can be collected, although some is relevant to particular problems.

The Diagnostics Collector is the recommended utility for collecting Java diagnostics files for a problem event. You can configure the JVM to run the Diagnostics Collector automatically, or you can manually run the utility after an event occurs. The Diagnostics Collector searches for dumps and log files, and produces a single compressed file containing all the diagnostic output for the problem event. For more information, see "The Diagnostics Collector" on page 328.

The data that is collected from a fault situation in z/OS depends on the problem symptoms, but could include some or all of the following information:

- Transaction dump - an unformatted dump that is requested by the MVS BCP IEATDUMP service. For more information, see "Setting up dumps" on page 176. This dump can be post-processed with the dump viewer (see "Using system dumps and the dump viewer" on page 271), the dbx debugger, or IPCS (Interactive Problem Control System).
- CEEDUMP - formatted application level dump, requested by the cdump system call.
- Javadump - formatted internal state data that is produced by the IBM JVM.
- Binary or formatted trace data from the JVM internal high performance trace. See "Using method trace" on page 316 and "Tracing Java applications and the JVM" on page 288.
- Debugging messages that are written to stderr. For example, the output from the JVM when switches like **-verbose:gc**, **-verbose**, or **-Xtgc** are used.
- Java stack traces when exceptions are thrown.
- Other unformatted system dumps obtained from middleware products or components (for example, SVC dumps requested by WebSphere for z/OS).
- SVC dumps obtained by the MVS Console DUMP command (typically for loops or hangs, or when the IEATDUMP fails).
- Trace data from other products or components (for example LE traces or the Component trace for z/OS UNIX).

- Heapdumps, if generated automatically, are required for problem determination. You should also take a Heapdump if you have a memory or performance problem.

NLS problem determination

The JVM contains built-in support for different locales. This section provides an overview of locales, with the main focus on fonts and font management.

The topics are:

Overview of fonts

When you want to show text, either in SDK components (AWT or Swing), on the console or in any application, characters must be mapped to glyphs.

A glyph is an artistic representation of the character, in some typographical style, and is stored in the form of outlines or bitmaps. Glyphs might not correspond one-for-one with characters. For instance, an entire character sequence can be represented as a single glyph. Also, a single character can be represented by more than one glyph (for example, in Indic scripts).

A font is a set of glyphs. Each glyph is encoded in a particular encoding format, so that the character to glyph mapping can be done using the encoded value. Almost all of the available Java fonts are encoded in Unicode and provide universal mappings for all applications.

The most commonly available font types are TrueType and OpenType fonts.

Font specification properties

Specify fonts according to the following characteristics:

Font family
Font family is a group of several individual fonts that are related in appearance. For example: Times, Arial, and Helvetica.

Font style
Font style specifies that the font is displayed in various faces. For example: Normal, Italic, and Oblique

Font variant
Font variant determines whether the font is displayed in normal caps or in small caps. A particular font might contain only normal caps, only small caps, or both types of glyph.

Font weight
Font weight describes the boldness or the lightness of the glyph to be used.

Font size
Font size is used to modify the size of the displayed text.

Fonts installed in the system

On Linux or UNIX platforms
To see the fonts that are either installed in the system or available for an application to use, type the command:

```
xset -q ""
```

If your **PATH** also points to the SDK (as expected), a result of running the command:

```
xset -q
```

is a list of the fonts that are bundled with the Developer Kit.

To add a font path, use the command:

```
xset +fp
```

To remove a font path, use the command:

```
xset -fp
```

Default font

If an application attempts to create a font that cannot be found, the font Dialog Lucida Sans Regular is used as the default font.

Font utilities

A list of font utilities that are supported.

Font utilities on AIX, Linux, and z/OS

xlsfonts
Use **xlsfonts** to check whether a particular font is installed on the system. For example: `xlsfonts | grep ksc` will list all the Korean fonts in the system.

iconv
Use to convert the character encoding from one encoding to other. Converted text is written to standard output. For example: `iconv -f oldset -t newset [file ...]`

Options are:

-f oldset
Specifies the source codeset (encoding).

-t newset
Specifies the destination codeset (encoding).

file
The file that contain the characters to be converted; if no file is specified, standard input is used.

Common NLS problem and possible causes

A common NLS problem with potential solutions.

Why do I see a square box or ??? (question marks) in the SDK components?
This effect is caused mainly because Java is not able to find the correct font file to display the character. If a Korean character should be displayed, the system should be using the Korean locale, so that Java can take the correct font file. If you are seeing boxes or queries, check the following items:

For Swing components:

1. Check your locale with `locale`
2. To change the locale, export `LANG=zh_TW` (for example)
3. If you know which font you have used in your application, such as serif, try to get the corresponding physical font by looking in the fontpath. If the font file is missing, try adding it there.

ORB problem determination

One of your first tasks when debugging an ORB problem is to determine whether the problem is in the client-side or in the server-side of the distributed application. Think of a typical RMI-IIOP session as a simple, synchronous communication between a client that is requesting access to an object, and a server that is providing it.

During this communication, a problem might occur in the execution of one of the following steps:

1. The client writes and sends a request to the server.
2. The server receives and reads the request.
3. The server executes the task in the request.
4. The server writes and sends a reply back.
5. The client receives and reads the reply.

It is not always easy to identify where the problem occurred. Often, the information that the application returns, in the form of stack traces or error messages, is not enough for you to make a decision. Also, because the client and server communicate through their ORBs, if a problem occurs, both sides will probably record an exception or unusual behavior.

This section describes all the clues that you can use to find the source of the ORB problem. It also describes a few common problems that occur more frequently. The topics are:

Identifying an ORB problem

A background of the constituents of the IBM ORB component.

What the ORB component contains

The ORB component contains the following items:

- Java ORB from IBM and rmi-iiop runtime environment (com.ibm.rmi.*, com.ibm.CORBA.*)
- RMI-IIOP API (javax.rmi.CORBA.*,org.omg.CORBA.*)
- IDL to Java implementation (org.omg.* and IBM versions com.ibm.org.omg.*)
- Transient name server (com.ibm.CosNaming.*, org.omg.CosNaming.*) - tnameserv
- -iiop and -idl generators (com.ibm.tools.rmi.rmic.*) for the rmic compiler - rmic
- idlj compiler (com.ibm.idl.*)

What the ORB component does not contain

The ORB component does *not* contain:

- RMI-JRMP (also known as Standard RMI)
- JNDI and its plug-ins

Therefore, if the problem is in java.rmi.* or sun.rmi.*, it is not an ORB problem. Similarly, if the problem is in com.sun.jndi.*, it is not an ORB problem.

Platform dependent problems

If possible, run the test case on more than one platform. All the ORB code is shared. You can nearly always reproduce genuine ORB problems on any platform. If you have a platform-specific problem, it is likely to be in some other component.

JIT problem

JIT bugs are very difficult to find. They might show themselves as ORB problems. When you are debugging or testing an ORB application, it is always safer to switch off the JIT by setting the option **-Xint**.

Fragmentation

Disable fragmentation when you are debugging the ORB. Although fragmentation does not add complications to the ORB's functioning, a fragmentation bug can be difficult to detect because it will most likely show as a general marshalling problem. The way to disable fragmentation is to set the ORB property **com.ibm.CORBA.FragmentSize=0**. You must do this on the client side and on the server side.

ORB versions

The ORB component carries a few version properties that you can display by calling the main method of the following classes:

1. `com.ibm.CORBA.iiop.Version` (ORB runtime version)
2. com.ibm.tools.rmic.iiop.Version (for tools; for example, **idlj** and **rmic**)
3. rmic -iiop -version (run the command line for **rmic**)

Limitation with bidirectional GIOP

Bidirectional GIOP is not supported.

Debug properties

Properties to use to enable ORB traces.

Attention: Do not enable tracing for normal operation, because it might cause performance degradation. Even if you have switched off tracing, FFDC (First Failure Data Capture) is still working, so that only serious errors are reported. If a debug file is produced, examine it to check on the problem. For example, the server might have stopped without performing an ORB.shutdown().

You can use the following properties to enable the ORB traces:

- **com.ibm.CORBA.Debug:**

Table 8. Debug property values

Property value	Trace output information
`false` [default]	No output
`fine`	Entry and exit points to the ORB code. Use this information to help identify the area of the ORB component that is causing the problem.
`finer`	As for `fine`, plus information about the working of ORB subcomponents. Use this information to help identify problems within a specific ORB subcomponent.
`finest` or `all`	Information about the entire ORB code flow

Note: If you use this property without specifying a value, full tracing is enabled.

- **com.ibm.CORBA.Debug.Output:** This property redirects traces to a file, which is known as a trace log. When this property is not specified, or it is set to an empty string, the file name defaults to the format orbtrc.DDMMYYYY.HHmm.SS.txt, where D=Day; M=Month; Y=Year; H=Hour (24 hour format); m=Minutes; S=Seconds. If the application (or Applet) does not have the privilege that it requires to write to a file, the trace entries go to stderr.
- **com.ibm.CORBA.CommTrace:** This property turns on wire tracing, also known as Comm tracing. Every incoming and outgoing GIOP message is sent to the trace log. You can set this property independently from Debug. This property is useful if you want to look only at the flow of information, and you are not interested in debugging the internals. The only two values that this property can have are **true** and **false**. The default is **false**.

Here is an example of common usage:

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

For rmic -iiop or rmic -idl, the following diagnostic tools are available:

- **-J-Djavac.dump.stack=1:** This tool ensures that all exceptions are caught.
- **-Xtrace:** This tool traces the progress of the parse step.

If you are working with an IBM SDK, you can obtain CommTrace for the transient name server (tnameserv) by using the standard environment variable **IBM_JAVA_OPTIONS**. In a separate command session to the server or client SDKs, you can use:

```
export IBM_JAVA_OPTIONS=-Dcom.ibm.CORBA.CommTrace=true -Dcom.ibm.CORBA.Debug=true
```

or the equivalent platform-specific command.

The setting of this environment variable affects each Java process that is started, so use this variable carefully. Alternatively, you can use the **-J** option to pass the properties through the tnameserv wrapper, as follows:

```
tnameserv -J-Dcom.ibm.CORBA.Debug=true
```

ORB exceptions

The exceptions that can be thrown are split into user and system categories.

If your problem is related to the ORB, unless your application is doing nothing or giving you the wrong result, your log file or terminal is probably full of exceptions that include the words “CORBA” and “rmi” many times. All unusual behavior that

occurs in a good application is highlighted by an exception. This principle also applies for the ORB with its CORBA exceptions. Similarly to Java, CORBA divides its exceptions into user exceptions and system exceptions.

User exceptions

User exceptions are IDL defined and inherit from org.omg.CORBA.UserException. These exceptions are mapped to checked exceptions in Java; that is, if a remote method raises one of them, the application that called that method must catch the exception. User exceptions are usually not unrecoverable exceptions and should always be handled by the application. Therefore, if you get one of these user exceptions, you know where the problem is, because the application developer had to make allowance for such an exception to occur. In most of these cases, the ORB is not the source of the problem.

System exceptions

System exceptions are thrown transparently to the application and represent an unusual condition in which the ORB cannot recover gracefully, such as when a connection is dropped. The CORBA 2.6 specification defines 31 system exceptions and their mapping to Java. They all belong to the org.omg.CORBA package. The CORBA specification defines the meaning of these exceptions and describes the conditions in which they are thrown.

The most common system exceptions are:

- **BAD_OPERATION:** This exception is thrown when an object reference denotes an existing object, but the object does not support the operation that was called.
- **BAD_PARAM:** This exception is thrown when a parameter that is passed to a call is out of range or otherwise considered not valid. An ORB might raise this exception if null values or null pointers are passed to an operation.
- **COMM_FAILURE:** This exception is raised if communication is lost while an operation is in progress, after the request was sent by the client, but before the reply from the server has been returned to the client.
- **DATA_CONVERSION:** This exception is raised if an ORB cannot convert the marshaled representation of data into its native representation, or cannot convert the native representation of data into its marshaled representation. For example, this exception can be raised if wide character codeset conversion fails, or if an ORB cannot convert floating point values between different representations.
- **MARSHAL:** This exception indicates that the request or reply from the network is structurally not valid. This error typically indicates a bug in either the client-side or server-side runtime. For example, if a reply from the server indicates that the message contains 1000 bytes, but the actual message is shorter or longer than 1000 bytes, the ORB raises this exception.
- **NO_IMPLEMENT:** This exception indicates that although the operation that was called exists (it has an IDL definition), no implementation exists for that operation.
- **UNKNOWN:** This exception is raised if an implementation throws a non-CORBA exception, such as an exception that is specific to the implementation's programming language. It is also raised if the server returns a system exception that is unknown to the client. If the server uses a later version of CORBA than the version that the client is using, and new system exceptions have been added to the later version this exception can happen.

Completion status and minor codes

Two pieces of data are associated with each system exception, these are described in this section.

- A completion status, which is an enumerated type that has three values: COMPLETED_YES, COMPLETED_NO and COMPLETED_MAYBE. These values indicate either that the operation was executed in full, that the operation was not executed, or that the execution state cannot be determined.
- A long integer, called minor code, that can be set to some ORB vendor-specific value. CORBA also specifies the value of many minor codes.

Usually the completion status is not very useful. However, the minor code can be essential when the stack trace is missing. In many cases, the minor code identifies the exact location of the ORB code where the exception is thrown and can be used by the vendor's service team to localize the problem quickly. However, for standard CORBA minor codes, this is not always possible. For example:

```
org.omg.CORBA.OBJECT_NOT_EXIST: SERVANT_NOT_FOUND   minor code: 4942FC11  completed: No
```

Minor codes are usually expressed in hexadecimal notation (except for Oracle's minor codes, which are in decimal notation) that represents four bytes. The OMG organization has assigned to each vendor a range of 4096 minor codes. The IBM vendor-specific minor code range is 0x4942F000 through 0x4942FFFF. “CORBA minor codes” on page 467 gives diagnostic information for common minor codes.

System exceptions might also contain a string that describes the exception and other useful information. You will see this string when you interpret the stack trace.

The ORB tends to map all Java exceptions to CORBA exceptions. A runtime exception is mapped to a CORBA system exception, while a checked exception is mapped to a CORBA user exception.

More exceptions other than the CORBA exceptions could be generated by the ORB component in a code bug. All the Java unchecked exceptions and errors and others that are related to the ORB tools rmic and idlj must be considered. In this case, the only way to determine whether the problem is in the ORB, is to look at the generated stack trace and see whether the objects involved belong to ORB packages.

Java security permissions for the ORB

When running with a Java SecurityManager, invocation of some methods in the CORBA API classes might cause permission checks to be made that could result in a SecurityException.

The following table shows methods affected when running with Java 2 SecurityManager:

Class/Interface	Method	Required permission
org.omg.CORBA.ORB	init	java.net.SocketPermission resolve
org.omg.CORBA.ORB	connect	java.net.SocketPermission listen

Class/Interface	Method	Required permission
org.omg.CORBA.ORB	resolve_initial_references	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_is_a	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_non_existent	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	OutputStream _request (String, boolean)	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_get_interface_def	java.net.SocketPermission connect
org.omg.CORBA. Request	invoke	java.net.SocketPermission connect
org.omg.CORBA. Request	send_deferred	java.net.SocketPermission connect
org.omg.CORBA. Request	send_oneway	java.net.SocketPermission connect
javax.rmi. PortableRemoteObject	narrow	java.net.SocketPermission connect

If your program uses any of these methods, ensure that it is granted the necessary permissions.

Interpreting the stack trace

Whether the ORB is part of a middleware application or you are using a Java stand-alone application (or even an applet), you must retrieve the stack trace that is generated at the moment of failure. It could be in a log file, or in your terminal or browser window, and it could consist of several chunks of stack traces.

The following example describes a stack trace that was generated by a server ORB running in the WebSphere Application Server:

```
org.omg.CORBA.MARSHAL: com.ibm.ws.pmi.server.DataDescriptor; IllegalAccessException  minor code: 4942F23E
completed: No
         at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:199)
         at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
         at com.ibm.rmi.io.ValueHandlerImpl.read_Array(ValueHandlerImpl.java:625)
         at com.ibm.rmi.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:273)
         at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:189)
         at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
         at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie._invoke(_EJSRemoteStatelessPmiService_
Tie.java:613)
         at com.ibm.CORBA.iiop.ExtendedServerDelegate.dispatch(ExtendedServerDelegate.java:515)
         at com.ibm.CORBA.iiop.ORB.process(ORB.java:2377)
         at com.ibm.CORBA.iiop.OrbWorker.run(OrbWorker.java:186)
         at com.ibm.ejs.oa.pool.ThreadPool$PooledWorker.run(ThreadPool.java:104)
         at com.ibm.ws.util.CachedThread.run(ThreadPool.java:137)
```

In the example, the ORB mapped a Java exception to a CORBA exception. This exception is sent back to the client later as part of a reply message. The client ORB reads this exception from the reply. It maps it to a Java exception (java.rmi.RemoteException according to the CORBA specification) and throws this new exception back to the client application.

Along this chain of events, often the original exception becomes hidden or lost, as does its stack trace. On early versions of the ORB (for example, 1.2.x, 1.3.0) the only way to get the original exception stack trace was to set some ORB debugging properties. Newer versions have built-in mechanisms by which all the nested stack traces are either recorded or copied around in a message string. When dealing with an old ORB release (1.3.0 and earlier), it is a good idea to test the problem on newer versions. Either the problem is not reproducible (known bug already solved) or the debugging information that you obtain is much more useful.

Description string:

The example stack trace shows that the application has caught a CORBA org.omg.CORBA.MARSHAL system exception. After the MARSHAL exception, some extra information is provided in the form of a string. This string should specify minor code, completion status, and other information that is related to the problem. Because CORBA system exceptions are alarm bells for an unusual condition, they also hide inside what the real exception was.

Usually, the type of the exception is written in the message string of the CORBA exception. The trace shows that the application was reading a value (read_value()) when an IllegalAccessException occurred that was associated to class com.ibm.ws.pmi.server.DataDescriptor. This information is an indication of the real problem and should be investigated first.

Interpreting ORB traces

The ORB trace file contains messages, trace points, and wire tracing. This section describes the various types of trace.

ORB trace message:

An example of an ORB trace message.

Here is a simple example of a message:

```
19:12:36.306 com.ibm.rmi.util.Version logVersions:110 P=754534:O=0:CT
ORBRas[default] IBM Java ORB build orbdev-20050927
```

This message records the time, the package, and the method name that was called. In this case, logVersions() prints out, to the log file, the version of the running ORB.

After the first colon in the example message, the line number in the source code where that method invocation is done is written (110 in this case). Next follows the letter P that is associated with the process number that was running at that moment. This number is related (by a hash) to the time at which the ORB class was loaded in that process. It is unlikely that two different processes load their ORBs at the same time.

The following O=0 (alphabetic O = numeric 0) indicates that the current instance of the ORB is the first one (number 0). CT specifies that this is the main (control) thread. Other values are: LT for listener thread, RT for reader thread, and WT for worker thread.

The ORBRas field shows which RAS implementation the ORB is running. It is possible that when the ORB runs inside another application (such as a WebSphere application), the ORB RAS default code is replaced by an external implementation.

The remaining information is specific to the method that has been logged while executing. In this case, the method is a utility method that logs the version of the ORB.

This example of a possible message shows the logging of entry or exit point of methods, such as:

```
14:54:14.848 com.ibm.rmi.iiop.Connection <init>:504 LT=0:P=650241:O=0:port=1360 ORBRas[default] Entry
.....
14:54:14.857 com.ibm.rmi.iiop.Connection <init>:539 LT=0:P=650241:O=0:port=1360 ORBRas[default] Exit
```

In this case, the constructor (that is, <init>) of the class Connection is called. The tracing records when it started and when it finished. For operations that include the java.net package, the ORBRas logger prints also the number of the local port that was involved.

Comm traces:

An example of comm (wire) tracing.

Here is an example of comm tracing:

```
 // Summary of the message containing name-value pairs for the principal fields
OUT GOING:
Request Message // It is an out going request, therefore we are dealing with a client
Date:           31 January 2003 16:17:34 GMT
Thread Info:    P=852270:O=0:CT
Local Port:     4899 (0x1323)
Local IP:       9.20.178.136
Remote Port:    4893 (0x131D)
Remote IP:      9.20.178.136
GIOP Version:   1.2
Byte order:     big endian

Fragment to follow: No     // This is the last fragment of the request
Message size:   276 (0x114)
--

Request ID:         5  // Request Ids are in ascending sequence
Response Flag:      WITH_TARGET // it means we are expecting a reply to this request
Target Address:     0
```

```
Object Key:         length = 26 (0x1A)  // the object key is created by the server when exporting
                                        // the servant and retrieved in the IOR using a naming service
                    4C4D4249 00000010 14F94CA4 00100000
                    00080000 00000000 0000
Operation:          message // That is the name of the method that the client invokes on the servant
Service Context:    length = 3 (0x3)   // There are three service contexts
Context ID:         1229081874 (0x49424D12) // Partner version service context. IBM only
Context data:       length = 8 (0x8)
                    00000000 14000005


Context ID:         1 (0x1)  // Codeset CORBA service context
Context data:       length = 12 (0xC)
                    00000000 00010001 00010100

Context ID:         6 (0x6)  // Codebase CORBA service context
Context data:       length = 168 (0xA8)
                    00000000 00000028 49444C3A 6F6D672E
                    6F72672F 53656E64 696E6743 6F6E7465
                    78742F43 6F646542 6173653A 312E3000
                    00000001 00000000 0000006C 00010200
                    0000000D 392E3230 2E313738 2E313336
                    00001324 0000001A 4C4D4249 00000010
                    15074A96 00100000 00080000 00000000
                    00000000 00000002 00000001 00000018
                    00000000 00010001 00000001 00010020
                    00010100 00000000 49424D0A 00000008
                    00000000 14000005
Data Offset:        11c
// raw data that goes in the wire in numbered rows of 16 bytes and the corresponding ASCII
decoding
0000: 47494F50 01020000 00000114 00000005    GIOP............
0010: 03000000 00000000 0000001A 4C4D4249    ............LMBI
0020: 00000010 14F94CA4 00100000 00080000    ......L.........
0030: 00000000 00000000 00000008 6D657373    ............mess
0040: 61676500 00000003 49424D12 00000008    age.....IBM.....
0050: 00000000 14000005 00000001 0000000C    ................
0060: 00000000 00010001 00010100 00000006    ................
0070: 000000A8 00000000 00000028 49444C3A    ...........(IDL:
0080: 6F6D672E 6F72672F 53656E64 696E6743    omg.org/SendingC
0090: 6F6E7465 78742F43 6F646542 6173653A    ontext/CodeBase:
00A0: 312E3000 00000001 00000000 0000006C    1.0............l
00B0: 00010200 0000000D 392E3230 2E313738    ........9.20.178
00C0: 2E313336 00001324 0000001A 4C4D4249    .136...$....LMBI
00D0: 00000010 15074A96 00100000 00080000    ......J.........
00E0: 00000000 00000000 00000002 00000001    ................
00F0: 00000018 00000000 00010001 00000001    ................
0100: 00010020 00010100 00000000 49424D0A    ... ........IBM.
0110: 00000008 00000000 14000005 00000000    ................
```

Note: The italic comments that start with a double slash have been added for clarity; they are not part of the traces.

In this example trace, you can see a summary of the principal fields that are contained in the message, followed by the message itself as it goes in the wire. In the summary are several field name-value pairs. Each number is in hexadecimal notation.

For details of the structure of a GIOP message, see the CORBA specification, chapters 13 and 15: http://www.omg.org/cgi-bin/doc?formal/99-10-07.

Client or server:

From the first line of the summary of the message, you can identify whether the host to which this trace belongs is acting as a server or as a client. OUT GOING means that the message has been generated on the workstation where the trace was taken and is sent to the wire.

In a distributed-object application, a server is defined as the provider of the implementation of the remote object to which the client connects. In this work, however, the convention is that a client sends a request while the server sends back a reply. In this way, the same ORB can be client and server in different moments of the rmi-iiop session.

The trace shows that the message is an outgoing request. Therefore, this trace is a client trace, or at least part of the trace where the application acts as a client.

Time information and host names are reported in the header of the message.

The Request ID and the Operation ("message" in this case) fields can be very helpful when multiple threads and clients destroy the logical sequence of the traces.

The GIOP version field can be checked if different ORBs are deployed. If two different ORBs support different versions of GIOP, the ORB that is using the more recent version of GIOP should fall back to a common level. By checking that field, however, you can easily check whether the two ORBs speak the same language.

Service contexts:

The header also records three service contexts, each consisting of a context ID and context data.

A service context is extra information that is attached to the message for purposes that can be vendor-specific such as the IBM Partner version that is described in the IOR in "The ORB" on page 64.

Usually, a security implementation makes extensive use of these service contexts. Information about an access list, an authorization, encrypted IDs, and passwords could travel with the request inside a service context.

Some CORBA-defined service contexts are available. One of these is the Codeset.

In the example, the codeset context has `ID 1` and data `00000000 00010001 00010100`. Bytes 5 through 8 specify that characters that are used in the message are encoded in ASCII (00010001 is the code for ASCII). Bytes 9 through 12 instead are related to wide characters.

The default codeset is UTF8 as defined in the CORBA specification, although almost all Windows and UNIX platforms typically communicate through ASCII. i5/OS and Mainframes such as zSeries systems are based on the IBM EBCDIC encoding.

The other CORBA service context, which is present in the example, is the Codebase service context. It stores information about how to call back to the client to access resources in the client such as stubs, and class implementations of parameter objects that are serialized with the request.

Common problems

This section describes some of the problems that you might find.

ORB application hangs:

One of the worst conditions is when the client, or server, or both, hang. If a hang occurs, the most likely condition (and most difficult to solve) is a deadlock of threads. In this condition, it is important to know whether the workstation on which you are running has more than one CPU, and whether your CPU is using Simultaneous Multithreading (SMT).

A simple test that you can do is to keep only one CPU running, disable SMT, and see whether the problem disappears. If it does, you know that you must have a synchronization problem in the application.

Also, you must understand what the application is doing while it hangs. Is it waiting (low CPU usage), or it is looping forever (almost 100% CPU usage)? Most of the cases are a waiting problem.

You can, however, still identify two cases:

- Typical deadlock
- Standby condition while the application waits for a resource to arrive

An example of a standby condition is where the client sends a request to the server and stops while waiting for the reply. The default behavior of the ORB is to wait indefinitely.

You can set a couple of properties to avoid this condition:

- com.ibm.CORBA.LocateRequestTimeout
- com.ibm.CORBA.RequestTimeout

When the property com.ibm.CORBA.enableLocateRequest is set to true (the default is false), the ORB first sends a short message to the server to find the object that it needs to access. This first contact is the Locate Request. You must now set the LocateRequestTimeout to a value other than 0 (which is equivalent to infinity). A good value could be something around 5000 ms.

Also, set the `RequestTimeout` to a value other than 0. Because a reply to a request is often large, allow more time for the reply, such as 10,000 ms. These values are suggestions and might be too low for slow connections. When a request runs out of time, the client receives an explanatory CORBA exception.

When an application hangs, consider also another property that is called com.ibm.CORBA.FragmentTimeout. This property was introduced in IBM ORB 1.3.1, when the concept of fragmentation was implemented to increase performance. You can now split long messages into small chunks or fragments and send one after the other over the net. The ORB waits for 30 seconds (default value) for the next fragment before it throws an exception. If you set this property, you disable this timeout, and problems of waiting threads might occur.

If the problem seems to be a deadlock or hang, capture the Javadump information. After capturing the information, wait for a minute or so, and do it again. A comparison of the two snapshots shows whether any threads have changed state. For information about how to do this operation, see “Triggering a Javadump” on page 241.

In general, stop the application, enable the orb traces and restart the application. When the hang is reproduced, the partial traces that can be retrieved can be used by the IBM ORB service team to help understand where the problem is.

Starting the client before the server is running:

If the client is started before the server is running, an error occurs when you run the client.

An example of the error messages that are generated from this process.

This operation outputs:

```
(org.omg.CORBA.COMM_FAILURE)
Hello Client exception:
        org.omg.CORBA.COMM_FAILURE:minor code:1 completed:No
             at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:145)
             at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:77)
             at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
             at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:75)
             at com.ibm.rmi.corba.ClientDelegate.createRequest(ClientDelegate.java:440)
             at com.ibm.rmi.corba.ClientDelegate.is_a(ClientDelegate.java:571)
             at org.omg.CORBA.portable.ObjectImpl._is_a(ObjectImpl.java:74)
             at org.omg.CosNaming.NamingContextHelper.narrow(NamingContextHelper.java:58)
             com.sun.jndi.cosnaming.CNCtx.callResolve(CNCtx.java:327)
```

Client and server are running, but not naming service:

An example of the error messages that are generated from this process.

The output is:

```
Hello Client exception:Cannot connect to ORB
Javax.naming.CommunicationException:
        Cannot connect to ORB.Root exception is org.omg.CORBA.COMM_FAILURE  minor code:1 completed:No
               at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:145)
               at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:77)
               at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
               at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:75)
               at com.ibm.rmi.corba.ClientDelegate.createRequest(ClientDelegate.java:440)
               at com.ibm.rmi.corba.InitialNamingClient.resolve(InitialNamingClient.java:197)
               at com.ibm.rmi.corba.InitialNamingClient.cachedInitialReferences(InitialNamingClient.j
               at com.ibm.rmi.corba.InitialNamingClient.resolve_initial_references(InitialNamingClien
               at com.ibm.rmi.corba.ORB.resolve_initial_references(ORB.java:1269)
               .........
```

You must start the Java IDL name server before an application or applet starts that uses its naming service. Installation of the Java IDL product creates a script (Solaris: tnameserv) or executable file that starts the Java IDL name server.

Start the name server so that it runs in the background. If you do not specify otherwise, the name server listens on port 2809 for the bootstrap protocol that is used to implement the ORB resolve_initial_references() and list_initial_references() methods.

Specify a different port, for example, 1050, as follows:

```
tnameserv -ORBInitialPort 1050
```

Clients of the name server must be made aware of the new port number. Do this by setting the org.omg.CORBA.ORBInitialPort property to the new port number when you create the ORB object.

Running the client with MACHINE2 (client) unplugged from the network:

An example of the error messages that are generated when the client has been unplugged form the network.

Your output is:

```
(org.omg.CORBA.TRANSIENT CONNECT_FAILURE)

Hello Client exception:Problem contacting address:corbaloc:iiop:machine2:2809/NameService
javax.naming.CommunicationException:Problem contacting address:corbaloc:iiop:machine2:2809/N
      is org.omg.CORBA.TRANSIENT:CONNECT_FAILURE (1)minor code:4942F301 completed:No
             at com.ibm.CORBA.transport.TransportConnectionBase.connect(TransportConnectionBase.jav
             at com.ibm.rmi.transport.TCPTransport.getConnection(TCPTransport.java:178)
             at com.ibm.rmi.iiop.TransportManager.get(TransportManager.java:79)
             at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:131)
             at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
             at com.ibm.CORBA.iiop.ClientDelegate._createRequest(ClientDelegate.java:2096)
             at com.ibm.CORBA.iiop.ClientDelegate.createRequest(ClientDelegate.java:1264)
             at com.ibm.CORBA.iiop.ClientDelegate.createRequest(ClientDelegate.java:1177)
             at com.ibm.rmi.corba.InitialNamingClient.resolve(InitialNamingClient.java:252)
             at com.ibm.rmi.corba.InitialNamingClient.cachedInitialReferences(InitialNamingClient.j
             at com.ibm.rmi.corba.InitialNamingClient.resolve_initial_references(InitialNamingClien
             at com.ibm.rmi.corba.InitialReferenceClient.resolve_initial_references(InitialReferenc
             at com.ibm.rmi.corba.ORB.resolve_initial_references(ORB.java:3211)
             at com.ibm.rmi.iiop.ORB.resolve_initial_references(ORB.java:523)
             at com.ibm.CORBA.iiop.ORB.resolve_initial_references(ORB.java:2898)
             ..........
```

IBM ORB service: collecting data

This section describes how to collect data about ORB problems.

If after all these verifications, the problem is still present, collect at all nodes of the problem the following information:

- Operating system name and version.
- Output of `java -version`.
- Output of `java com.ibm.CORBA.iiop.Version`.
- Output of `rmic -iiop -version`, if rmic is involved.
- ASV build number (WebSphere Application Server only).
- If you think that the problem is a regression, include the version information for the most recent known working build and for the failing build.
- If this is a runtime problem, collect debug and communication traces of the failure from each node in the system (as explained earlier in this section).
- If the problem is in `rmic -iiop` or `rmic -idl`, set the options: `-J-Djavac.dump.stack=1 -Xtrace`, and capture the output.
- Typically this step is not necessary. If it looks like the problem is in the buffer fragmentation code, IBM service will return the defect asking for an additional set of traces, which you can produce by executing with `-Dcom.ibm.CORBA.FragmentSize=0`.

A testcase is not essential, initially. However, a working testcase that demonstrates the problem by using only the Java SDK classes will speed up the resolution time for the problem.

Preliminary tests:

The ORB is affected by problems with the underlying network, hardware, and JVM.

When a problem occurs, the ORB can throw an org.omg.CORBA.* exception, some text that describes the reason, a minor code, and a completion status. Before you assume that the ORB is the cause of problem, do the following checks:

- Check that the scenario can be reproduced in a similar configuration.
- Check that the JIT is disabled (see "JIT and AOT problem determination" on page 322).

Also:

- Disable additional CPUs.
- Disable Simultaneous Multithreading (SMT) where possible.
- Eliminate memory dependencies with the client or server. The lack of physical memory can be the cause of slow performance, apparent hangs, or crashes. To remove these problems, ensure that you have a reasonable headroom of memory.
- Check physical network problems (firewalls, comm links, routers, DNS name servers, and so on). These are the major causes of CORBA COMM_FAILURE exceptions. As a test, ping your own workstation name.
- If the application is using a database such as DB2, switch to the most reliable driver. For example, to isolate DB2 AppDriver, switch to Net Driver, which is slower and uses sockets, but is more reliable.

Attach API problem determination

This section helps you solve problems involving the Attach API.

The IBM Java Attach API uses shared semaphores, sockets, and file system artifacts to implement the attach protocol. Problems with these artifacts might adversely affect the operation of applications when they use the attach API.

Note: Error messages from agents on the target VM go to `stderr` or `stdout` for the target VM. They are not reported in the messages output by the attaching VM.

Deleting or changing permissions on directories and files in `/tmp`

The attach API depends on the contents of a common directory. By default the common directory is `/tmp/.com_ibm_tools_attach`, but you can specify a different directory by using the **`-Dcom.ibm.tools.attach.directory`** system property. The common directory must have owner, group, and world read, write, and execute permissions, and the sticky bit must be set. The common files `_attachlock`, `_master`, and `_notifier` must have owner, group, and world read and write permissions. Execute permissions are not required.

Problems are caused if you modify the common directory in one of the following ways:

- Deleting the common directory.
- Deleting the contents of the common directory.
- Changing the permissions of the common directory or any of its content.

If you do modify the common directory, possible effects include:

- Semaphore "leaks" might occur, where excessive numbers of unused shared semaphores are opened. You can remove the semaphores using the command:

  ```
  ipcrm -s <semid>
  ```

  Use the command to delete semaphores that have keys starting with "0xa1".
- The Java VMs might not be able to list existing target VMs.

- The Java VMs might not be able to attach to existing target VMs.
- The Java VM might not be able to enable its attach API.
- Java processes might not terminate, or might take an excessive length of time to terminate.

If the common directory cannot be used, a Java VM attempts to re-create the common directory. However, the JVM cannot re-create the files that are related to VMs that are currently running.

If the `/tmp` directory, or the directory specified by the **`-Dcom.ibm.tools.attach.directory`** system property, is full or inaccessible (for example, because of permissions), the Attach API fails to initialize and no error message is produced.

z/OS console messages reporting security violations in `/tmp`

The Attach API stores control files in the directory `/tmp/.com_ibm_tools_attach`. To prevent the display of security violation messages, use one of the following options:

- Add a security exception.
- Specify a different control directory, by setting the **`com.ibm.tools.attach.directory`** system property.

The VirtualMachine.attach(String id) method reports `AttachNotSupportedException: No provider for virtual machine id`

There are several possible reasons for this message:

- The target VM might be owned by another userid. The attach API can only connect a VM to a target VM with the same userid.
- The attach API for the target VM might not have launched yet. There is a short delay from when the Java VM launches to when the attach API is functional.
- The attach API for the target VM might have failed. Verify that the directory `/tmp/.com_ibm_tools_attach/<id>` exists, and that the directory is readable and writable by the userid.
- The target directory `/tmp/.com_ibm_tools_attach/<id>` might have been deleted.
- The attach API might not have been able to open the shared semaphore. To verify that there is at least one shared semaphore, use the command:

  ```
  ipcs -s
  ```

  If there is a shared semaphore, at least one key starting with "0xa1" appears in the output from the **`ipcs`** command.

  Note: The number of available semaphores is limited on systems which use System V IPC, including Linux, z/OS, and AIX.

The VirtualMachine.attach() method reports `AttachNotSupportedException`

There are several possible reasons for this message:

- The target process is dead or suspended.
- The target process, or the hosting system is heavily loaded. The result is a delay in responding to the attach request.

- The network protocol has imposed a wait time on the port used to attach to the target. The wait time might occur after heavy use of the attach API, or other protocols which use sockets. To check if any ports are in the `TIME_WAIT` state, use the command:

  ```
  netstat -a
  ```

The VirtualMachine.loadAgent(), VirtualMachine.loadAgentLibrary(),, or VirtualMachine.loadAgentPath() methods report `com.sun.tools.attach.AgentLoadException` or `com.sun.tools.attach.AgentInitializationException`

There are several possible reasons for this message:

- The JVMTI agent or the agent JAR file might be corrupted. Try loading the agent at startup time using the `-javaagent`, `-agentlib`, or `-agentpath` option, depending on which method reported the problem.
- The agent might be attempting an operation which is not available after VM startup.

A process running as root can see a target using AttachProvider.listVirtualMachines(), but attempting to attach results in an `AttachNotSupportedException`

A process can attach only to processes owned by the same user. To attach to a non-root process from a root process, first use the **su** command to change the effective UID of the attaching process to the UID of the target UID, before attempting to attach.

The `/tmp/.com_ibm_tools_attach` directory contains many directories with numeric file names

Each Java process creates a private directory in the `/tmp/.com_ibm_tools_attach` directory, using its process ID as the directory name. When the process exits, it deletes this private directory. If the process, or the operating system, crashes, or you end the process by using the **SIGKILL** command, these obsolete directories are not removed.

Subsequent Java processes delete obsolete directories automatically over time. Each process examines a sample of the directories in `/tmp/.com_ibm_tools_attach`. If a directory is obsolete and is owned by the user that is running the process, the process deletes that directory.

To force deletion of all obsolete directories that are owned by the current user, run the **jconsole** command, found in the SDK `bin` directory. When the New Connection dialog is displayed, click **Cancel**, then exit the application.

To clean up all obsolete directories for all users, run the **jconsole** command as the `root` user.

Related concepts:

"Support for the Java Attach API" on page 141
Your application can connect to another "target" virtual machine using the Java Attach API. Your application can then load an agent application into the target virtual machine, for example to perform tasks such as monitoring status.

Using diagnostic tools

Diagnostic tools are available to help you solve your problems.

This section describes how to use the tools. The chapters are:

- "Overview of the available diagnostic tools"
- "Using dump agents" on page 221
- "Using Javadump" on page 240
- "Using Heapdump" on page 262
- "Using system dumps and the dump viewer" on page 271
- "Tracing Java applications and the JVM" on page 288
- "JIT and AOT problem determination" on page 322
- "Garbage Collector diagnostic data" on page 333
- "Class-loader diagnostic data" on page 341
- "Shared classes diagnostic data" on page 344
- "Using the Reliability, Availability, and Serviceability Interface" on page 372
- "Using the HPROF Profiler" on page 385
- "Using the JVMTI" on page 389
- "Using the Diagnostic Tool Framework for Java" on page 405
- "Using JConsole" on page 412

Note: JVMPI is now a deprecated interface, replaced by JVMTI.

Overview of the available diagnostic tools

The diagnostic information that can be produced by the JVM is described in the following topics. A range of supplied tools can be used to post-process this information and help with problem determination.

Subsequent topics in this part of the *Information Center* give more details on the use of the information and tools in solving specific problem areas.

Some diagnostic information (such as that produced by Heapdump) is targeted towards specific areas of Java (classes and object instances in the case of Heapdumps), whereas other information (such as tracing) is targeted towards more general JVM problems.

Categorizing the problem

During problem determination, one of the first objectives is to identify the most probable area where the problem originates.

Many problems that seem to be a Java problem originate elsewhere. Areas where problems can arise include:

- The JVM itself
- Native code
- Java applications

- An operating system or system resource
- A subsystem (such as database code)
- Hardware

You might need different tools and different diagnostic information to solve problems in each area. The tools described here are (in the main) those built in to the JVM or supplied by IBM for use with the JVM. The majority of these tools are cross-platform tools, although there might be the occasional reference to other tools that apply only to a specific platform or varieties of that platform. Many other tools are supplied by hardware or system software vendors (such as system debuggers). Some of these tools are introduced in the platform-specific sections.

Summary of diagnostic information

A running IBM JVM includes mechanisms for producing different types of diagnostic data when different events occur.

In general, the production of this data happens under default conditions, but can be controlled by starting the JVM with specific options (such as **-Xdump**; see "Using dump agents" on page 221). Older versions of the IBM JVM controlled the production of diagnostic information through the use of environment variables. You can still use these environment variables, but they are not the preferred mechanism and are not discussed in detail here. "Environment variables" on page 469 lists the supported environment variables).

The format of the various types of diagnostic information produced is specific to the IBM JVM and might change between releases of the JVM.

The types of diagnostic information that can be produced are:

Javadump
: The Javadump is sometimes referred to as a Javacore or thread dump in some JVMs. This dump is in a human-readable format produced by default when the JVM terminates unexpectedly because of an operating system signal, an OutOfMemoryError exception, or when the user enters a reserved key combination (for example, **Ctrl-Break** on Windows). You can also generate a Javadump by calling a method from the Dump API, for example com.ibm.jvm.Dump.JavaDump(), from inside the application. A Javadump summarizes the state of the JVM at the instant the signal occurred. Much of the content of the Javadump is specific to the IBM JVM. See "Using Javadump" on page 240 for details.

Heapdump
: The JVM can generate a Heapdump at the request of the user (for example by calling com.ibm.jvm.Dump.HeapDump() from inside the application) or (by default) when the JVM terminates because of an OutOfMemoryError exception. You can specify finer control of the timing of a Heapdump with the **-Xdump:heap** option. For example, you could request a Heapdump after a certain number of full garbage collections have occurred. The default Heapdump format (phd files) is not human-readable and you process it using available tools such as Heaproots. See "Using Heapdump" on page 262 for more details.

System dumps
: System dumps (also known as core dumps on Linux platforms) are platform-specific files that contain information about the active processes, threads, and system memory. System dumps are usually large. By default, system dumps are produced by the JVM only when the JVM fails

unexpectedly because of a GPF (general protection fault) or a major JVM or system error. You can also request a system dump by using the Dump API. For example, you can call the com.ibm.jvm.Dump.SystemDump() method from your application. You can use the **-Xdump:system** option to produce system dumps when other events occur.

Garbage collection data
A JVM started with the **-verbose:gc** option produces output in XML format that can be used to analyze problems in the Garbage Collector itself or problems in the design of user applications. Numerous other options affect the nature and amount of Garbage Collector diagnostic information produced. See "Garbage Collector diagnostic data" on page 333 for more information.

Trace data
The IBM JVM tracing allows execution points in the Java code and the internal JVM code to be logged. The **-Xtrace** option allows the number and areas of trace points to be controlled, as well as the size and nature of the trace buffers maintained. The internal trace buffers at a time of failure are also available in a system dump and tools are available to extract them from a system dump. Generally, trace data is written to a file in an encoded format and then a trace formatter converts the data into a readable format. However, if small amounts of trace are to be produced and performance is not an issue, trace can be routed to STDERR and will be pre-formatted. For more information, see "Tracing Java applications and the JVM" on page 288.

Other data
Special options are available for producing diagnostic information relating to

- The JIT (see "JIT and AOT problem determination" on page 322)
- Class loading (see "Class-loader diagnostic data" on page 341)
- Shared classes (see "Shared classes diagnostic data" on page 344)

You can also download the IBM Monitoring and Diagnostic Tools for Java, a set of freely-available GUI-based tools for monitoring Java applications and analyzing diagnostic data. For more information, see "IBM Monitoring and Diagnostic Tools for Java" on page 192.

Summary of cross-platform tooling

IBM has several cross-platform diagnostic tools. The following sections provide brief descriptions of the tools and indicate the different areas of problem determination to which they are suited.

IBM Monitoring and Diagnostic Tools for Java:

The IBM Monitoring and Diagnostic Tools for Java are a set of GUI-based tools for monitoring Java applications and analyzing diagnostic data. These tools are designed to make Java diagnostic tasks as quick and as easy as possible.

Some tools can be attached to a running JVM, to monitor application behavior and resource usage. For other tools, you generate dump files from your system or JVM, then analyze the file in the tool. By using the tools, you can diagnose problems such as memory leaks, thread contention issues, and I/O bottlenecks, as well as getting information and recommendations to help you tune the JVM and improve the performance of your application.

For more information about the tools, see “Using the IBM Monitoring and Diagnostic Tools for Java” on page 219.

Cross-platform dump viewer:

The cross-system dump viewer uses the dump files that the operating system generates to resolve data relevant to the JVM.

This tool is provided in two parts:

1. **jextract** - platform-specific utility to extract and package (compress) data from the dump generated by the native operating system. This part of the process is required only for system dumps that have been generated from earlier versions of the JVM.
2. **jdmpview** - a cross-platform Java tool to view that data

The dump viewer “understands” the JVM and can be used to analyze its internals. It is a useful tool to debug unexpected terminations of the JVM. The tool is provided only in the IBM SDK for Java. Because the dump viewer is cross-platform, you can analyze a dump from any system, and without knowledge of the system debugger.

For more information, see “Using system dumps and the dump viewer” on page 271.

JVMTI tools:

The JVMTI (JVM Tool Interface) is a programming interface for use by tools. It replaces the Java Virtual Machine Profiler Interface (JVMPI) and the Java Virtual Machine Debug Interface (JVMDI).

For information on the JVMTI, see “Using the JVMTI” on page 389. The HPROF tool provided with the SDK has been updated to use the JVMTI; see “Using the HPROF Profiler” on page 385.

JVMPI tools:

JVMPI is no longer available; you must upgrade existing tools to use the JVMTI (Java Virtual Machine Tool Interface), described in “Using the JVMTI” on page 389. An article to help you with the upgrade is at:

http://www.oracle.com/technetwork/articles/javase/jvmpitransition-138768.html

The IBM SDK provided tool HPROF has been updated to use the JVMTI; see “Using the HPROF Profiler” on page 385.

JPDA tools:

The Java Platform Debugging Architecture (JPDA) is a common standard for debugging JVMs. The IBM Virtual Machine for Java is fully JPDA compatible.

Any JPDA debugger can be attached to the IBM Virtual Machine for Java. Because they are debuggers, JPDA tools are best suited to tracing application problems that have repeatable conditions, such as:

- Memory leaks in applications.
- Unexpected termination or “hanging”.

An example of a JPDA tool is the debugger that is bundled with Eclipse for Java.

DTFJ:

The Diagnostic Tool Framework for Java (DTFJ) is a Java application programming interface (API) from IBM used to support the building of Java diagnostics tools.

DTFJ can examine a system dump to analyze the internal structure of the JVM.

DTFJ is implemented in pure Java and tools written using DTFJ can be cross-platform. Therefore, it is possible to analyze a dump taken from one machine on another (remote and more convenient) machine. For example, a dump produced on an AIX PPC machine can be analyzed on a Windows Thinkpad.

For more information, see "Using the Diagnostic Tool Framework for Java" on page 405.

Trace formatting:

JVM trace is a key diagnostic tool for the JVM. The IBM JVM incorporates a large degree of flexibility in determining what is traced and when it is traced. This flexibility enables you to tailor trace so that it has a relatively small effect on performance.

The IBM Virtual Machine for Java contains many embedded trace points. **In this release, maximal tracing is enabled by default for a few level 1 tracepoints and exception trace points.** Command-line options allow you to set exactly what is to be traced, and specify where the trace output is to go. Trace output is generally in an encoded format and requires a trace formatter to be viewed successfully.

In addition to the embedded trace points provided in the JVM code, you can place your own application trace points in your Java code. You can activate tracing for entry and exit against all methods in all classes. Alternatively, you can activate tracing for a selection of methods in a selection of classes. Application and method traces are interleaved in the trace buffers with the JVM embedded trace points. The tracing allows detailed analysis of the routes taken through the code.

Tracing is used mainly for performance and leak problem determination. Trace data might also provide clues to the state of a JVM before an unexpected termination or "hang".

Trace and trace formatting are IBM-specific; that is, they are present only in the IBM Virtual Machine for Java. See "Using method trace" on page 316 and "Tracing Java applications and the JVM" on page 288 for more details. Although trace is not easy to understand, it is an effective tool.

JVMRI:

The JVMRI interface will be deprecated in the future and replaced by JVMTI extensions.

The JVMRI (JVM RAS Interface, where RAS stands for Reliability, Availability, Serviceability) allows you to control several JVM operations programmatically.

For example, the IBM Virtual Machine for Java contains a large number of embedded trace points. Most of these trace points are switched off by default. A

JVMRI agent can act as a Plug-in to allow real-time control of trace information. You use the **-Xrun** command-line option so that the JVM itself loads the agent at startup. When loaded, a JVMRI agent can dynamically switch individual JVM trace points on and off, control the trace level, and capture the trace output.

The JVMRI is particularly useful when applied to performance and leak problem determination, although the trace file might provide clues to the state of a JVM before an unexpected termination or hang.

The RAS Plug-in interface is an IBM-specific interface; that is, it is present only in the IBM Virtual Machine for Java. See "Using the Reliability, Availability, and Serviceability Interface" on page 372 for details. You need some programming skills and tools to be able to use this interface.

Scenarios in which dumps might not be produced

In certain scenarios, a dump is not produced when a crash occurs. This section gives reasons why a dump is not produced and suggests how you can obtain a system dump.

A crash can occur with no dump produced. An example scenario is one in which the crash occurs during the shut down of the Java runtime environment. The Java runtime environment might not have time to produce all the debug information. In this case, the console output shows the start of the dump information, but the Java runtime environment cannot write the information in a dump file. For example, the console might show the following output:

```
Unhandled exception
Type=Segmentation error vmState=0x00000000
J9Generic_Signal_Number=00000004 ExceptionCode=c0000005 ExceptionAddress=430514B
E ContextFlags=0001003f
Handler1=7FEE9C40 Handler2=7FEC98C0 InaccessibleAddress=00000000
EDI=000A7060 ESI=43159598 EAX=00000000 EBX=001925EC
ECX=00000001 EDX=4368FECC
EIP=430514BE ESP=4368FED4 EBP=4368FED8 EFLAGS=00010246
Module=failing_module.dll
Module_base_address=43050000 Offset_in_DLL=000014be
Target=2_40_20081203_026494_lHdSMr (Windows XP 5.1 build 2600 Service Pack 2)
CPU=x86 (2 logical CPUs) (0x7fe6b000 RAM)
```

A diagnostic dump is not produced for several possible reasons. A common reason is that the Java runtime process was stopped by a user, a script, or by the operating system. Another possible reason is that the crash occurred on a JVM process that was very close to shut down, resulting in a race condition between the JVM dump handler and the main thread exiting the process.

Identifying if the race condition exists:

Enable trace points to check for situations in which no dump is produced after a crash.

About this task

Check for the situations in which no dump is produced after a crash by enabling trace points near the shut down of the Java runtime environment. If the trace points overlap with the crash condition, you have confirmation that the race condition occurred. The tracepoints in the protectedDestroyJavaVM are the last to be triggered before the main thread returns.

Procedure

1. Find the protectedDestroyJavaVM function tracepoints in the `J9TraceFormat.dat` file by using the instructions in "Determining the tracepoint ID of a tracepoint" on page 312.
2. When you have the tracepoint IDs, rerun the failing scenario with those tracepoints sent to the console. The results are similar to the following output:

```
java -Xtrace:print=tpnid{j9vm.381-394} MyApp

11:10:09.421*0x42cc1a00            j9vm.385      > protectedDestroyJavaVM
11:10:09.421 0x42cc1a00            j9vm.386      - protectedDestroyJavaVM waiting for Java threads to
stop
11:10:09.421 0x42cc1a00            j9vm.387      - protectedDestoryJavaVM all Java threads have stopped
11:10:09.421 0x42cc1a00            j9vm.388      - protectedDestroyJavaVM protectedDestroyJavaVM
vmCleanup complete
11:10:09.421 0x42cc1a00            j9vm.389      - protectedDestroyJavaVM VM Shutting Down Hook Fired
Unhandled exception
Type=Segmentation error vmState=0x00000000

J9Generic_Signal_Number=00000004 ExceptionCode=c0000005 ExceptionAddress=430514BE ContextFlags=0001003f
Handler1=7FEE9C40 Handler2=7FEC98C0 InaccessibleAddress=00000000
EDI=000A70A0 ESI=432235D8 EAX=00000000 EBX=00192684
ECX=00000001 EDX=4368FECC
EIP=430514BE ESP=4368FED4 EBP=4368FED8 EFLAGS=00010246
Module=failing_module.dll
Module_base_address=43050000 Offset_in_DLL=000014be
11:10:09.421 0x42cc1a00            j9vm.390      - Target=2_40_20081203_026494_lHdSMr (Windows XP 5.1
build 2600 Service Pack 2)
protectedDestroyJavaVM GC HeapManagement ShutdownCPU=x86 (2 logical CPUs) (0x7fe6b000 RAM)

11:10:09.421 0x42cc1a00            j9vm.391      - protectedDestroyJavaVM vmShutdown returned
11:10:09.421 0x42cc1a00            j9vm.393      - protectedDestroyJavaVM terminateRemainingThreads failed
```

The `Unhandled exception` message is printed after the first tracepoints for the protectedDestroyJavaVM function. This output shows that the crash occurred very late in the life of the Java runtime environment, and that enough time remained to produce the dumps before the process ended.

What to do next

When you confirm that a race condition has occurred, you might still be able to obtain a system dump. For more information, see "Obtaining system dumps in a race condition."

Obtaining system dumps in a race condition:

You might be able to obtain system dumps even when a race condition exists.

About this task

When you confirm that you have a race condition in which shut down timing prevents a system dump, you can try to obtain a dump in two ways:

- Try to prevent the system from shutting down before the dump is taken.
- Add a delay near the end of the JVM run time to give the dump handler enough time to write the dumps.

Procedure

On AIX, z/OS, or Linux, create a system dump by using the **-Xrs** Java command-line option to disable the Java signal handler. The default signal handler

in the operating system triggers a dump and prevents the system from shutting down before the dump is taken. For more information, see "Disabling dump agents with **-Xrs** and **-Xrs:sync**" on page 239.

Using the IBM Monitoring and Diagnostic Tools for Java

The IBM Monitoring and Diagnostic Tools for Java are a set of GUI-based tools which you can use to monitor your Java applications, analyze resource usage, and diagnose problems. The tools can help you to optimize application performance, improve application stability, reduce resource usage, and resolve problems more quickly.

The tools provide output in various formats, such as tables, charts, graphs, and recommendations. Use this output to complete the following diagnostic tasks:

- Detect deadlock conditions
- Monitor thread activity
- See which methods are taking the most time to run
- See which objects are using the most memory
- Find memory leaks and I/O bottlenecks
- Analyze the efficiency of Java collections, such as arrays
- Understand the relationships between application objects
- Visualize garbage collection performance
- Get recommendations for tuning the JVM and improving application performance

The following tools are available:

Health Center
Monitor a running JVM, with minimal performance overhead. Tuning recommendations are also provided.

Garbage Collection and Memory Visualizer
Analyze the memory usage, garbage collection behavior, and performance of Java applications, by plotting verbose garbage collection data from dump files. Tuning recommendations are also provided.

Interactive Diagnostic Data Explorer
Use commands to extract information from dump files. This tool is a GUI-based version of the **jdmpview** command, with extra features.

Memory Analyzer
Analyze the memory usage and performance of Java applications, using data from dump files.

The tools are available to download, free of charge, into the IBM Support Assistant. The IBM Support Assistant is a free workbench that is designed to help you with problem determination. The IBM Monitoring and Diagnostic Tools for Java is just one set of tools that you can install into the IBM Support Assistant.

For more information about the IBM Monitoring and Diagnostic Tools for Java, see IBM Monitoring and Diagnostic Tools for Java information center and IBM Monitoring and Diagnostic Tools for Java developerWorks page.

Garbage Collection and Memory Visualizer

Garbage Collection and Memory Visualizer (GCMV) helps you understand memory use, garbage collection behavior, and performance of Java applications.

GCMV parses and plots data from various types of log, including the following types:

- Verbose garbage collection logs.
- Trace garbage collection logs, generated by using the -Xtgc parameter.
- Native memory logs, generated by using the ps, **svmon**, or **perfmon** system commands.

The tool helps to diagnose problems such as memory leaks, analyze data in various visual formats, and provides tuning recommendations.

GCMV is provided as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see: http://www.ibm.com/developerworks/java/jdk/tools/gcmv/.

Further information about GCMV is available in an IBM Information Center.

Health Center

Health Center is a diagnostic tool for monitoring the status of a running Java Virtual Machine (JVM).

The tool is provided in two parts:

- The Health Center agent that collects data from a running application.
- An Eclipse-based client that connects to the agent. The client interprets the data and provides recommendations to improve the performance of the monitored application.

Health Center is provided as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see: http://www.ibm.com/developerworks/java/jdk/tools/healthcenter/.

Further information about Health Center is available in an IBM Information Center.

Interactive Diagnostic Data Explorer

Interactive Diagnostic Data Explorer (IDDE) is a GUI-based alternative to the dump viewer (**jdmpview** command). IDDE provides the same functionality as the dump viewer, but with extra support such as the ability to save command output.

Use IDDE to more easily explore and examine dump files that are produced by the JVM. Within IDDE, you enter commands in an investigation log, to explore the dump file. The support that is provided by the investigation log includes the following items:

- Command assistance
- Auto-completion of text, and some parameters such as class names
- The ability to save commands and output, which you can then send to other people
- Highlighted text and flagging of issues
- The ability to add your own comments
- Support for using the Memory Analyzer from within IDDE

IDDE is provided as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see IDDE overview on developerWorks.

Further information about IDDE is available in an IBM Information Center.

Memory Analyzer

Memory Analyzer helps you analyze Java heaps using operating system level dumps and Portable Heap Dumps (PHD).

This tool can analyze dumps that contain millions of objects, providing the following information:

- The retained sizes of objects.
- Processes that are preventing the Garbage Collector from collecting objects.
- A report to automatically extract leak suspects.

This tool is based on the Eclipse Memory Analyzer (MAT) project, and uses the IBM Diagnostic Tool Framework for Java (DTFJ) feature to enable the processing of dumps from IBM JVMs.

Memory Analyzer is provided as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see: http://www.ibm.com/developerworks/java/jdk/tools/memoryanalyzer/.

Further information about Memory Analyzer is available in an IBM Information Center.

Using dump agents

Dump agents are set up during JVM initialization. They enable you to use events occurring in the JVM, such as Garbage Collection, thread start, or JVM termination, to initiate dumps or to start an external tool.

The default dump agents are sufficient for most cases. Use the **-Xdump** option to add and remove dump agents for various JVM events, update default dump settings (such as the dump name), and limit the number of dumps that are produced.

This section describes:

Using the -Xdump option

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

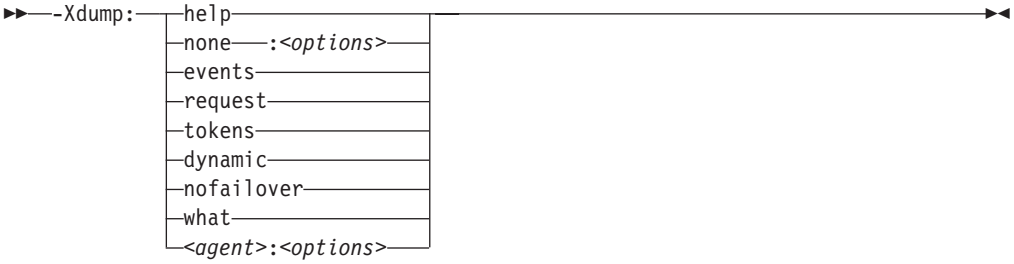
You can use the **-Xdump** option to:

- 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:

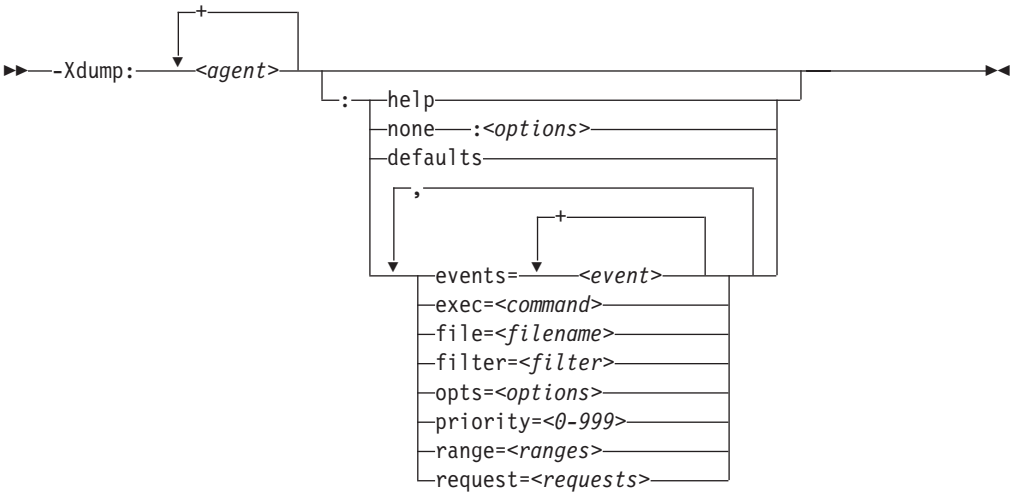
```
java -Xdump:heap:none -Xdump:heap+java:events=vmstart+vmstop <class> [args...]
```

You can use the **-Xdump: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



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

Merging -Xdump agents:

-Xdump agents are always merged internally by the JVM, as long as none of the agent settings conflict with each other.

If you configure more than one dump agent, each responds to events according to its configuration. However, the internal structures representing the dump agent configuration might not match the command line, because dump agents are merged for efficiency. Two sets of options can be merged as long as none of the agent settings conflict. This means that the list of installed dump agents and their parameters produced by **-Xdump:what** might not be grouped in the same way as the original **-Xdump** options that configured them.

For example, you can use the following command to specify that a dump agent collects a javadump on class unload:

```
java -Xdump:java:events=unload -Xdump:what
```

This command does not create a new agent, as can be seen in the results from the **-Xdump:what** option.

```
...
----------------------
-Xdump:java:
    events=gpf+user+abort+unload,
    label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
    range=1..0,
    priority=10,
    request=exclusive
----------------------
```

The configuration is merged with the existing javadump agent for events **gpf**, **user**, and **abort**, because none of the specified options for the new unload agent conflict with those for the existing agent.

In the previous example, if one of the parameters for the unload agent is changed so that it conflicts with the existing agent, then it cannot be merged. For example, the following command specifies a different priority, forcing a separate agent to be created:

```
java -Xdump:java:events=unload,priority=100 -Xdump:what
```

The results of the **-Xdump:what** option in the command are as follows.

```
...
----------------------
-Xdump:java:
    events=unload,
    label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
    range=1..0,
    priority=100,
    request=exclusive
----------------------
-Xdump:java:
    events=gpf+user+abort,
    label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
    range=1..0,
    priority=10,
    request=exclusive
----------------------
```

To merge dump agents, the **request**, **filter**, **opts**, **label**, and **range** parameters must match exactly. If you specify multiple agents that filter on the same string, but keep all other parameters the same, the agents are merged. For example:

```
java -Xdump:none -Xdump:java:events=uncaught,filter=java/lang/NullPointerException \\
-Xdump:java:events=unload,filter=java/lang/NullPointerException -Xdump:what
```

The results of this command are as follows.

```
Registered dump agents
----------------------
-Xdump:java:
    events=unload+uncaught,
    filter=java/lang/NullPointerException,
    label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
    range=1..0,
    priority=10,
    request=exclusive
----------------------
```

Dump agents

A dump agent performs diagnostic tasks when triggered. Most dump agents save information on the state of the JVM for later analysis. The "tool" agent can be used to trigger interactive diagnostic data.

The following table shows the dump agents:

Dump agent	Description
stack	Stack dumps are very basic dumps in which the status and Java stack of the thread is written to stderr. See "Stack dumps" on page 225.
console	Basic thread dump to stderr.
system	Capture raw process image. See "Using system dumps and the dump viewer" on page 271.
tool	Run command-line program.
java	Write application summary. See "Using Javadump" on page 240.
heap	Capture heap graph. See "Using Heapdump" on page 262.
snap	Take a snap of the trace buffers.
ceedump	**(z/OS only)** Produce an LE CEEDUMP.

Console dumps:

Console dumps are very basic dumps, in which the status of every Java thread is written to stderr.

In this example, the **range=1..1** suboption is used to control the amount of output to just one thread start (in this case, the start of the Signal Dispatcher thread).

```
java -Xdump:console:events=thrstart+thrstop,range=1..1

JVMDUMP006I Processing Dump Event "thrstart", detail "" - Please Wait.
-------- Console dump --------

Stack Traces of Threads:


ThreadName=main(08055B18)
Status=Running

ThreadName=JIT Compilation Thread(08056038)
Status=Waiting
Monitor=08055914 (JIT-CompilationQueueMonitor)
Count=0
Owner=(00000000)

^^^^^^^^ Console dump ^^^^^^^^
JVMDUMP013I Processed Dump Event "thrstart", detail "".
```

Two threads are displayed in the dump because the main thread does not generate a thrstart event.

System dumps:

System dumps involve dumping the address space and as such are generally very large.

The bigger the footprint of an application the bigger its dump. A dump of a major server-based application might take up many gigabytes of file space and take several minutes to complete. In this example, the file name is overridden from the default.

```
java -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
```

See “Using system dumps and the dump viewer” on page 271 for more information about analyzing a system dump.

Stack dumps:

Stack dumps are very basic dumps in which the status and Java stack of the thread is written to stderr. Stack dumps are very useful when used together with the "allocation" dump event to identify Java code that is allocating large objects.

In the following example, the `main` thread has allocated a byte array of size 1549128 bytes:

```
JVMDUMP006I Processing dump event "allocation", detail "1549128 bytes, type byte[]" - please wait.
Thread=main (0188701C) Status=Running
        at sun/misc/Resource.getBytes()[B (Resource.java:109)
        at java/net/URLClassLoader.defineClass(Ljava/lang/String;Lsun/misc/Resource;)Ljava/lang/Class;
(URLClassLoader.java:489)
        at java/net/URLClassLoader.access$300(Ljava/net/URLClassLoader;Ljava/lang/String;Lsun/misc/
Resource;)Ljava/lang/Class; (URLClassLoader.java:64)
        at java/net/URLClassLoader$ClassFinder.run()Ljava/lang/Object; (URLClassLoader.java:901)
        at java/security/AccessController.doPrivileged(Ljava/security/PrivilegedExceptionAction;Ljava/
security/AccessControlContext;)Ljava/lang/Object; (AccessController.java:284)
        at java/net/URLClassLoader.findClass(Ljava/lang/String;)Ljava/lang/Class; (URLClassLoader.
java:414)
        at java/lang/ClassLoader.loadClass(Ljava/lang/String;Z)Ljava/lang/Class; (ClassLoader.java:643)
        at sun/misc/Launcher$AppClassLoader.loadClass(Ljava/lang/String;Z)Ljava/lang/Class; (Launcher.
java:345)
        at java/lang/ClassLoader.loadClass(Ljava/lang/String;)Ljava/lang/Class; (ClassLoader.java:609)
        at TestLargeAllocations.main([Ljava/lang/String;)V (TestLargeAllocations.java:49)
```

LE CEEDUMPs:

LE CEEDUMPs are a z/OS only formatted summary system dump that show stack traces for each thread that is in the JVM process, together with register information and a short dump of storage for each register.

This example of a traceback is taken from a CEEDUMP produced by a crash. The traceback shows that the crash occurred in the rasTriggerMethod method:

```
CEE3DMP V1 R8.0: CHAMBER.JVM.TDUMP.CHAMBER.D080910.T171047                    09/10/08 5:10:52 PM                Page:    4

  Traceback:
    DSA Addr  Program Unit  PU Addr   PU Offset  Entry         E Addr    E  Offset   Statement  Load Mod  Service  Status
    .........
    124222A8  CEEHDSP       07310AF0  +00000CEC  CEEHDSP       07310AF0  +00000CEC              CEEPLPKA  UK34253  Call
    12421728  CEEHRNUH      0731F728  +00000092  CEEHRNUH      0731F728  +00000092              CEEPLPKA  HLE7730  Call
    128461E0                12AB6A00  +0000024C  rasTriggerMethod
                                                               12AB6A00  +0000024C        2120  *PATHNAM  j080625  Exception
    12846280                12AACBE8  +00000208  hookMethodEnter
                                                               12AACBE8  +00000208        1473  *PATHNAM  j080625  Call
    12846300                12A547C0  +000000B8  J9HookDispatch
                                                               12A547C0  +000000B8         157  *PATHNAM  j080625  Call
    12846380                12943840  +00000038  triggerMethodEnterEvent
                                                               12943840  +00000038         110  *PATHNAM  j080625  Call
    .........
```

When a CEEDUMP is produced by the JVM, the following message is issued:

```
JVMDUMP010I CEE dump written to /u/test/CEEDUMP.20090622.133914.65649
```

On 32-bit z/OS, if more than one CEEDUMP is produced during the lifetime of a JVM instance, the second and subsequent CEEDUMPs will be appended to the same file. The `JVMDUMP010I` messages will identify the same file each time.

On 64-bit z/OS, if more than one CEEDUMP is produced a separate CEEDUMP file is written each time, and the `JVMDUMP010I` messages will identify the separate files.

The CEEDUMP is not produced by default. Use the **ceedump** dump agent to enable CEEDUMP production, for example:

```
java -Xdump:ceedump:events=gpf
```

See Understanding the Language Environment dump in the *z/OS: Language Environment Debugging Guide* for more information.

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**. Alternatively, see the topic "Dump agent tokens" on page 234. If you do not specify a tool to use, a platform-specific debugger is started.

```
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 is started only once. To start the tool every time the event occurs, set the **range** option to 1..0. For more information, see "range option" on page 232.

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.

Javadumps:

Javadumps are an internally generated and formatted analysis of the JVM, giving information that includes the Java threads present, the classes loaded, and heap statistics.

An example of producing a Javadump when a class is loaded:

```
java -Xdump:java:events=load,filter=java/lang/String -version

 JVMDUMP006I Processing dump event "load", detail "java/lang/String" - please wait.
 JVMDUMP007I JVM Requesting Java dump using '/home/user/javacore.20090602.094449.274632.0001.txt'
 JVMDUMP010I Java dump written to /home/user/javacore.20090602.094449.274632.0001.txt
 JVMDUMP013I Processed dump event "load", detail "java/lang/String".
```

See "Using Javadump" on page 240 for more information about analyzing a Javadump.

Heapdumps:

Heapdumps produce phd format files by default.

"Using Heapdump" on page 262 provides more information about Heapdumps. The following example shows the production of a Heapdump. In this case, both a phd and a classic (.txt) Heapdump have been requested by the use of the **opts=** option.

```
java -Xdump:heap:events=vmstop,opts=PHD+CLASSIC -version

JVMDUMP006I Processing dump event "vmstop", detail "#00000000" - please wait.
JVMDUMP007I JVM Requesting Heap dump using '/home/user/heapdump.20090602.095239.164050.0001.phd'
JVMDUMP010I Heap dump written to /home/user/heapdump.20090602.095239.164050.0001.phd
JVMDUMP007I JVM Requesting Heap dump using '/home/user/heapdump.20090602.095239.164050.0001.txt'
JVMDUMP010I Heap dump written to /home/user/heapdump.20090602.095239.164050.0001.txt
JVMDUMP013I Processed dump event "vmstop", detail "#00000000".
```

See “Using Heapdump” on page 262 for more information about analyzing a Heapdump.

Snap traces:

Snap traces are controlled by **-Xdump**. They contain the tracepoint data held in the trace buffers.

The following example shows the production of a snap trace.

```
java -Xdump:snap:events=vmstop -version

 JVMDUMP006I Processing dump event "vmstop", detail "#00000000" - please wait.
 JVMDUMP007I JVM Requesting Snap dump using '/home/user/Snap.20090603.063646.315586.0001.trc'
 JVMDUMP010I Snap dump written to /home/user/Snap.20090603.063646.315586.0001.trc
 JVMDUMP013I Processed dump event "vmstop", detail "#00000000".
```

Snap traces require the use of the trace formatter for further analysis.

See “Using the trace formatter” on page 311 for more information about analyzing a snap trace.

Dump events

Dump agents are triggered by events occurring during JVM operation.

Some events can be filtered to improve the relevance of the output. See “filter option” on page 230 for more information.

Note: The gpf and abort events cannot trigger a heap dump, prepare the heap (request=prepwalk), or compact the heap (request=compact).

The following table shows events available as dump agent triggers:

Event	Triggered when...	Filter operation
gpf	A General Protection Fault (GPF) occurs.	
user	The JVM receives the SIGQUIT (Linux, AIX, z/OS, and i5/OS) or SIGBREAK (Windows) signal from the operating system.	
abort	The JVM receives the SIGABRT signal from the operating system.	
vmstart	The virtual machine is started.	
vmstop	The virtual machine stops.	Filters on exit code; for example, **filter=#129..#192#-42#255**
load	A class is loaded.	Filters on class name; for example, **filter=java/lang/String**
unload	A class is unloaded.	
throw	An exception is thrown.	Filters on exception class name; for example, **filter=java/lang/OutOfMem***
catch	An exception is caught.	Filters on exception class name; for example, **filter=*Memory***
uncaught	A Java exception is not caught by the application.	Filters on exception class name; for example, **filter=*MemoryError**

Event	Triggered when...	Filter operation
systhrow	A Java exception is about to be thrown by the JVM. This is different from the 'throw' event because it is only triggered for error conditions detected internally in the JVM.	Filters on exception class name; for example, **filter=java/lang/OutOfMem***
thrstart	A new thread is started.	
blocked	A thread becomes blocked.	
thrstop	A thread stops.	
fullgc	A garbage collection cycle is started.	
slow	A thread takes longer than 50ms to respond to an internal JVM request.	Changes the time taken for an event to be considered slow; for example, **filter=#300ms** will trigger when a thread takes longer than 300ms to respond to an internal JVM request.
allocation	A Java object is allocated with a size matching the given filter specification	Filters on object size; a filter must be supplied. For example, **filter=#5m** will trigger on objects larger than 5 Mb. Ranges are also supported; for example, **filter=#256k..512k** will trigger on objects between 256 Kb and 512 Kb in size.
traceassert	An internal error occurs in the JVM	Not applicable.
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.

Advanced control of dump agents

Options are available to give you more control over dump agent behavior.

exec option:

The exec option is used by the tool dump agent to specify an external application to start.

See "Tool option" on page 227 for an example and usage information.

file option:

The **file** option is used by dump agents that write to a file.

The **file** option specifies where the diagnostics information is written. For example:

```
java -Xdump:heap:events=vmstop,file=my.dmp
```

When producing system dumps or CEEDUMPs on z/OS platforms, use the **dsn** option instead of the **file** option. For example:

```
java -Xdump:system:events=vmstop,dsn=%uid.MYDUMP
```

You can use tokens to add context to dump file names. See "Dump agent tokens" on page 234 for more information.

The location for the dump is selected from these options, in this order:

1. The location specified on the command line.
2. The location specified by the relevant environment variable.

- **_CEE_DMPTARG** for Javadump.
- **_CEE_DMPTARG** for Heapdump.
- **JAVA_DUMP_TDUMP_PATTERN** for system dumps.
- **_CEE_DMPTARG** for snap traces.

3. The current working directory of the JVM process.

If the directory does not exist, it is created.

If the dump cannot be written to the selected location, the JVM reverts to using the following locations, in this order:

1. The location specified by the **TMPDIR** environment variable.
2. The `/tmp` directory.

This JVM action does not apply to CEEDUMPs on z/OS platforms that use the **dsn** option.

You can prevent the JVM reverting to different dump locations by using the **-Xdump:nofailover** option.

filter option:

Some JVM events occur thousands of times during the lifetime of an application. Dump agents can use filters and ranges to avoid excessive dumps being produced.

Wildcards

You can use a wildcard in your exception event filter by placing an asterisk only at the beginning or end of the filter. The following command does not work because the second asterisk is not at the end:

```
-Xdump:java:events=throw,filter=*InvalidArgumentException#*.myVirtualMethod
```

In order to make this filter work, it must be changed to:

```
-Xdump:java:events=throw,filter=*InvalidArgumentException#MyApplication.*
```

Class loading and exception events

You can filter class loading (load) and exception (throw, catch, uncaught, systhrow) events by Java class name:

```
-Xdump:java:events=throw,filter=java/lang/OutOfMem*
-Xdump:java:events=throw,filter=*MemoryError
-Xdump:java:events=throw,filter=*Memory*
```

You can filter throw, uncaught, and systhrow exception events by Java method name:

```
-Xdump:java:events=throw,filter=ExceptionClassName[#ThrowingClassName.
throwingMethodName[#stackFrameOffset]]
```

Optional portions are shown in brackets.

You can filter the catch exception events by Java method name:

```
-Xdump:java:events=catch,filter=ExceptionClassName[#CatchingClassName.
catchingMethodName]
```

Optional portions are shown in brackets.

vmstop event

You can filter the JVM shut down event by using one or more exit codes:

```
-Xdump:java:events=vmstop,filter=#129..192#-42#255
```

slow event

You can filter the slow event to change the time threshold from the default of 50 ms:

```
-Xdump:java:events=slow,filter=#300ms
```

You cannot set the filter to a time that is less than the default time.

Other events

If you apply a filter to an event that does not support filtering, the filter is ignored.

opts option:

The Heapdump agent uses this option to specify the type of file to produce. On z/OS, the system dump agent uses this option to specify the type of dump to produce.

Heapdumps and the opts option

You can specify a PHD Heapdump, a classic text Heapdump, or both. For example:

–Xdump:heap:opts=PHD (default)
–Xdump:heap:opts=CLASSIC
–Xdump:heap:opts=PHD+CLASSIC

For more information, see “Enabling text formatted ("classic") Heapdumps” on page 263.

z/OS System dumps and the opts option

You can specify a system transaction dump (IEATDUMP), an LE dump (CEEDUMP), or both. For example:

–Xdump:system:opts=IEATDUMP (default)
–Xdump:system:opts=CEEDUMP
–Xdump:system:opts=IEATDUMP+CEEDUMP

The ceedump agent is the preferred way to specify LE dumps, for example:

```
-Xdump:ceedump:events=gpf
```

Tool dumps and the opts option

The tool dump agent supports two options that can be specified using the `opts` option. You can run the external process asynchronously with `opts=ASYNC`. You can also specify a delay in milliseconds that produces a pause after starting the command. These two options can be used independently or together. The following examples show different options for starting a new process that runs `myProgram`:

```
-Xdump:tool:events=vmstop,exec=myProgram
```

Without the opts option, the tool dump agent starts the process, and waits for the process to end before continuing.

```
-Xdump:tool:events=vmstop,exec=myProgram,opts=ASYNC
```

When opts=ASYNC is specified, the tool dump agent starts the process, and continues without waiting for the new process to end.

```
-Xdump:tool:events=vmstop,exec=myProgram,opts=WAIT1000
```

This option starts the process, waits for the process to end, and then waits a further 1 second (1000 milliseconds) before continuing.

```
-Xdump:tool:events=vmstop,exec=myProgram,opts=ASYNC+WAIT10000
```

Finally the last example starts the process and waits for 10 seconds before continuing, whether the process is still running or not. This last form is useful if you are starting a process that does not end, but requires time to initialize properly.

For more information about using the dump agent tool option, see "Tool option" on page 227.

priority option:

One event can generate multiple dumps. The agents that produce each dump run sequentially and their order is determined by the **priority** keyword set for each agent.

Examination of the output from **-Xdump:what** shows that a gpf event produces a snap trace, a Javadump, and a system dump. In this example, the system dump runs first, with priority 999. The snap dump runs second, with priority 500. The Javadump runs last, with priority 10:

```
–Xdump:heap:events=vmstop,priority=123
```

The maximum value allowed for **priority** is 999. Higher priority dump agents are started first.

If you do not specifically set a priority, default values are taken based on the dump type. The default priority and the other default values for a particular type of dump, can be displayed by using **-Xdump:<*type*>:defaults**. For example:

```
java -Xdump:heap:defaults -version

Default -Xdump:heap settings:

  events=gpf+user
  filter=
  file=/home/user/heapdump.%Y%m%d.%H%M%S.%pid.phd
  range=1..0
  priority=40
  request=exclusive+prepwalk
  opts=PHD
```

range option:

You can start and stop dump agents on a particular occurrence of a JVM event by using the range suboption.

For example:

```
-Xdump:java:events=fullgc,range=100..200
```

Note: **range=1..0** against an event means "on every occurrence".

The JVM default dump agents have the **range** option set to `1..0` for all events except `systhrow`. All `systhrow` events with `filter=java/lang/OutOfMemoryError` have the **range** set to `1..4`, which limits the number of dumps produced on `OutOfMemory` conditions to a maximum of 4. For more information, see "Default dump agents" on page 234

If you add a new dump agent and do not specify the range, a default of `1..0` is used.

request option:

Use the request option to ask the JVM to prepare the state before starting the dump agent.

The available options are listed in the following table:

Option value	Description
exclusive	Request exclusive access to the JVM.
compact	Run garbage collection. This option removes all unreachable objects from the heap before the dump is generated.
prepwalk	Prepare the heap for walking. You must also specify **exclusive** when using this option.
serial	Suspend other dumps until this one has finished.
preempt	Applies to the Java dump agent and controls whether native threads in the process are forcibly pre-empted in order to collect stack traces. If this option is not specified, only Java stack traces are collected in the Javadump.

For example, the default setting of the request option for javadumps is `request=exclusive+preempt`. To change the settings so that javadumps are produced without pre-empting threads to collect native stack traces, use the following option:

```
-Xdump:java:request=exclusive
```

In general, the default request options are sufficient.

You can specify more than one request option using +. For example:

```
-Xdump:heap:request=exclusive+compact+prepwalk
```

defaults option:

Each dump type has default options. To view the default options for a particular dump type, use **-Xdump:<type>:defaults**.

You can change the default options at run time. For example, you can direct Java dump files into a separate directory for each process, and guarantee unique files by adding a sequence number to the file name using:

```
-Xdump:java:defaults:file=dumps/%pid/javacore-%seq.txt
```

Or, for example, on z/OS, you can add the jobname to the Java dump file name using:

```
-Xdump:java:defaults:file=javacore.%job.%H%M%S.txt
```

This option does not add a Javadump agent; it updates the default settings for Javadump agents. Further Javadump agents will then create dump files using this specification for filenames, unless overridden.

Note: Changing the defaults for a dump type will also affect the default agents for that dump type added by the JVM during initialization. For example if you change the default file name for Javadumps, that will change the file name used by the default Javadump agents. However, changing the default **range** option will not change the range used by the default Javadump agents, because those agents override the **range** option with specific values.

Dump agent tokens

Use tokens to add context to dump file names and to pass command-line arguments to the tool agent.

The tokens available are listed in the following table:

Token	Description
%Y	Year (4 digits)
%y	Year (2 digits)
%m	Month (2 digits)
%d	Day of the month (2 digits)
%H	Hour (2 digits)
%M	Minute (2 digits)
%S	Second (2 digits)
%pid	Process id
%uid	User name
%seq	Dump counter
%tick	msec counter
%home	Java home directory
%last	Last dump
%job	Job name (z/OS only)
&DS	Dump Section. An incrementing sequence number used for splitting TDUMP files to be less than 2 GB in size. (z/OS 64-bit version 1.10 or newer only)

Default dump agents

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

See “Removing dump agents” on page 235 for more information.

Use the **-Xdump:what** option on the command line to show the registered dump agents. The sample output shows the default dump agents that are in place:

```
java -Xdump:what

Registered dump agents
----------------------
-Xdump:system:
```

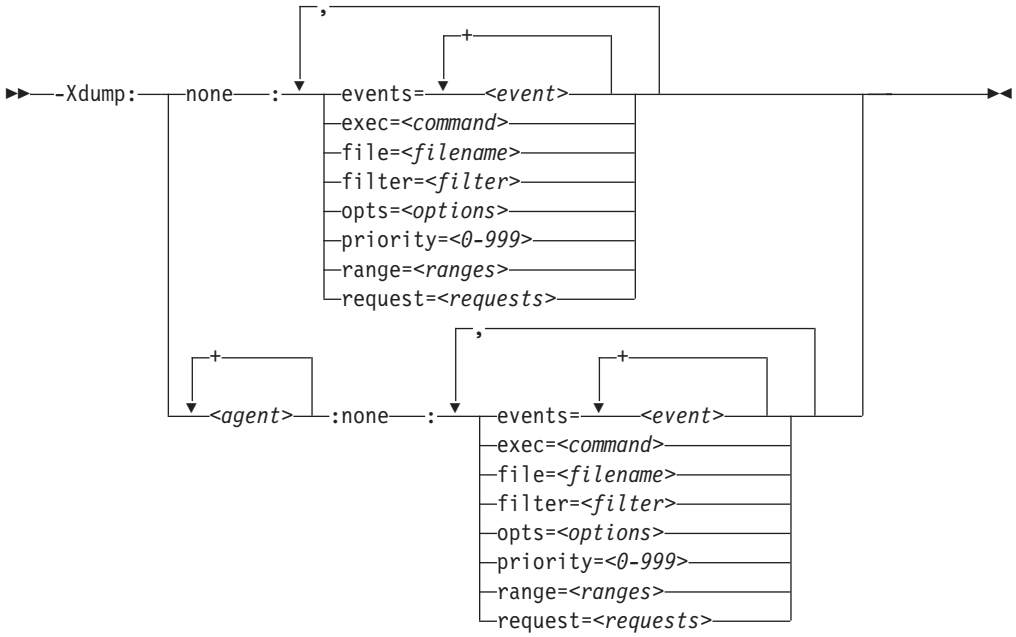
```
    events=gpf+user+abort+traceassert+corruptcache,
    label=%uid.JVM.TDUMP.%job.D%y%m%d.T%H%M%S,
    range=1..0,
    priority=999,
    request=serial
----------------------
-Xdump:system:
    events=systhrow,
    filter=java/lang/OutOfMemoryError,
    label=%uid.JVM.TDUMP.%job.D%y%m%d.T%H%M%S,
    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
----------------------
```

Removing dump agents

You can remove all default dump agents and any preceding dump options by using **-Xdump:none**.

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

-Xdump command-line syntax: the none option



Use this option so that you can subsequently specify a completely new dump configuration.

You can also remove dump agents of a particular type. 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 "Specifying command-line options" on page 417 for more information on using a command-line options file.

Dump agent environment variables

The **-Xdump** option on the command line is the preferred method for producing dumps for cases where the default settings are not enough. You can also produce dumps using the **JAVA_DUMP_OPTS** environment variable.

If you set agents for a condition using the **JAVA_DUMP_OPTS** environment variable, default dump agents for that condition are disabled; however, any **-Xdump** options specified on the command line will be used.

The **JAVA_DUMP_OPTS** environment variable is used as follows:

```
JAVA_DUMP_OPTS="ON<condition>(<agent>[<count>],<agent>[<count>]),
ON<condition>(<agent>[<count>],...),...)"
```

where:

- <*condition*> can be:
  - ANYSIGNAL
  - DUMP
  - ERROR
  - INTERRUPT
  - EXCEPTION
  - OUTOFMEMORY
- <*agent*> can be:
  - ALL
  - NONE
  - JAVADUMP
  - SYSDUMP
  - HEAPDUMP
  - CEEDUMP (z/OS specific)
- <*count*> is the number of times to run the specified agent for the specified condition. This value is optional. By default, the agent will run every time the condition occurs.

JAVA_DUMP_OPTS is parsed by taking the leftmost occurrence of each condition, so duplicates are ignored. The following setting will produce a system dump for the first error condition only:

```
ONERROR(SYSDUMP[1]),ONERROR(JAVADUMP)
```

Also, the ONANYSIGNAL condition is parsed before all others, so

```
ONINTERRUPT(NONE),ONANYSIGNAL(SYSDUMP)
```

has the same effect as

```
ONANYSIGNAL(SYSDUMP),ONINTERRUPT(NONE)
```

If the **JAVA_DUMP_TOOL** environment variable is set, that variable is assumed to specify a valid executable name and is parsed for replaceable fields, such as %pid. If %pid is detected in the string, the string is replaced with the JVM's own process

ID. The tool specified by **JAVA_DUMP_TOOL** is run after any system dump or Heapdump has been taken, before anything else.

Other environments variables available for controlling dumps are listed in "Javadump and Heapdump options" on page 471.

The dump settings are applied in the following order, with the settings later in the list taking precedence:

1. Default JVM dump behavior.
2. **-Xdump** command-line options that specify **-Xdump:<type>:defaults**, see "defaults option" on page 233.
3. **DISABLE_JAVADUMP**, **IBM_HEAPDUMP**, and **IBM_HEAP_DUMP** environment variables.
4. **IBM_JAVADUMP_OUTOFMEMORY** and **IBM_HEAPDUMP_OUTOFMEMORY** environment variables.
5. **JAVA_DUMP_OPTS** environment variable.
6. Remaining **-Xdump** command-line options.

Setting **JAVA_DUMP_OPTS** only affects those conditions that you specify. Actions on other conditions are unchanged.

Signal mappings

The signals used in the **JAVA_DUMP_OPTS** environment variable map to multiple operating system signals.

When setting the **JAVA_DUMP_OPTS** environment variable, the mapping of operating system signals to the "condition" is as follows:

	z/OS	Windows	Linux, AIX, and i5/OS
EXCEPTION	SIGTRAP		SIGTRAP
	SIGILL	SIGILL	SIGILL
	SIGSEGV	SIGSEGV	SISEGV
	SIGFPE	SIGFPE	SIGFPE
	SIGBUS		SIGBUS
	SIGSYS		
	SIGXFSZ		SIGXFSZ
INTERRUPT	SIGINT	SIGINT	SIGINT
	SIGTERM	SIGTERM	SIGTERM
	SIGHUP		SIGHUP
ERROR	SIGABRT	SIGABRT	SIGABRT
DUMP	SIGQUIT		SIGQUIT
		SIGBREAK	

Using dump agents on z/OS

Dump output is written to different files, depending on the type of the dump. File names include a time stamp. The z/OS platform has an additional dump type called CEEDUMP.

The CEEDUMP is not produced by default. Use the `ceedump` dump agent to enable CEEDUMP production.

If **CEEDUMP** is specified, an LE CEEDUMP is produced for the relevant conditions, after any system dump processing, but before a Javadump is produced. A CEEDUMP is a formatted summary system dump that shows stack traces for each thread that is in the JVM process, together with register information and a short dump of storage for each register.

On z/OS, you can change the behavior of LE by setting the **_CEE_RUNOPTS** environment variable. See the *LE Programming Reference* for more information. In particular, the **TRAP** option determines whether LE condition handling is enabled, which, in turn, drives JVM signal handling, and the **TERMTHDACT** option indicates the level of diagnostic information that LE should produce.

For more information about **CEEDUMP** see "LE CEEDUMPs" on page 226

On 64-bit z/OS, TDUMP files are split into several smaller files if the TDUMP exceeds the 2 GB file size limit. Each file is given a sequence number. If you specify a template for the TDUMP file name, each instance of the **&DS** parameter is replaced in the actual file name by an ordered sequence number. For example, **X&DS** generates file names in the form `X01`, `X02`, `X03` and so on. If you specify a template but omit the **&DS** parameter, it is appended automatically to the end of the template. If you do not specify a template, the default template is used, and **.X&DS** is appended to the end of the template. If the resulting template exceeds the maximum length allowed for a TDUMP data set name, a message is issued and catalogued in NLS, advising that the template pattern is too long to append *.X&DS*, and that a default pattern will be used: `%uid.JVM.%job.D%y%m%d.T%H%M%S.X&DS`

To merge the sequence of TDUMP files, use the TSO panel IPCS->utility->copy MVS dump data set. If you have copied or moved the IEATDUMP files from MVS to the USS file system, you can use the **cat** command to merge the files. For example:

```
cat JVM.TDUMP.X001 JVM.TDUMP.X002 > JVM.TDUMP.FULL
```

Dump filenames and locations

Dump files produced on z/OS include:

- **Sytem dump**: On TSO as a standard MVS data set, using the default name of the form: `%uid.JVM.TDUMP.%job.D%Y%m%d.T%H%M%S` (31-bit), `%uid.JVM.%job.D%y%m%d.T%H%M%S.X&DS` (64-bit), or as determined by the setting of the **JAVA_DUMP_TDUMP_PATTERN** environment variable.
- **LE CEEDUMP**: In the directory specified by **_CEE_DMPTARG**, or the current directory if **_CEE_DMPTARG** is not specified, using the file name: `CEEDUMP.%Y%m%d.%H%M%S.%pid`.
- **Heapdump**: In the current directory as a file named `heapdump.%Y%m%d.T%H%M%S.phd`. See "Using Heapdump" on page 262 for more information.
- **Javadump**: In the same directory as CEEDUMP, or standard Javadump directory as: `javacore.%Y%m%d.%H%M%S.%pid.%seq.txt`.

Disabling dump agents with -Xrs and -Xrs:sync

When using a debugger such as **GDB** or **WinDbg** to diagnose problems in JNI code, you might want to disable the signal handler of the Java runtime environment so that any signals received are handled by the operating system.

Using the **-Xrs** command-line option prevents the Java runtime environment handling exception signals such SIGSEGV and SIGABRT. When the Java runtime

signal handler is disabled, a SIGSEGV or GPF crash does not call the JVM dump agents. Instead, dumps are produced depending on the operating system.

For more information about the **-Xrs** and **-Xrs:sync** options, see "JVM command-line options" on page 428.

Disabling dump agents in z/OS

The behavior on z/OS depends on system configuration. By default, a message is printed on the syslog:

```
N 0000000 MVW0     08323 13:19:27.59 STC05748 00000010 IEF450I ANDHALL2 *OMVSEX -
ABEND=S0C4 U0000 REASON=00000004
```

A message is also displayed on the Java process `stderr`:

```
CEE3204S The system detected a protection exception (System Completion Code=0C4)
.
          From entry point rasTriggerMethod at compile unit offset +000000001265A
344 at entry offset +00000000000002CC at address 000000001265A344.
[1] + Done(139) J6.0_64/bin/java -Xrs -Xtrace:trigger=Method{*.main,segv} TestApp
   83951806      Segmentation violation  J6.0_64/bin/java
```

When you include **TERMTHDATA(UADUMP)** in the CEE runtime options, a CEEDUMP is written to the working directory. To collect a dump suitable for processing by **jextract** and **DTFJ**, set an appropriate SLIP trap to trigger a dump on the failure condition. For information about setting SLIP traps, see the MVS commands reference: http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/iea2g181/CONTENTS?SHELF=EZ2CMZ81.bks&DT=20080118081647#COVER

Using Javadump

Javadump produces files that contain diagnostic information that is related to the JVM and a Java application that is captured at a point during execution. For example, the information can be about the operating system, the application environment, threads, stacks, locks, and memory.

Javadumps are human readable and do not contain any Java object content or data, except for the following items:

- Thread names, with thread IDs and flags
- Classloader names, with counts and flags
- Class and method names
- Some heap addresses

The preferred way to control the production of Javadumps is by enabling dump agents using **–Xdump:java:** on application startup. See "Using dump agents" on page 221. You can also control Javadumps by the use of environment variables. See "Environment variables and Javadump" on page 262. Default agents are in place that create Javadumps when the JVM ends unexpectedly or when an out-of-memory exception occurs, unless the defaults are overridden. Javadumps are also triggered by default when specific signals are received by the JVM.

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.

This chapter describes:

- "Enabling a Javadump" on page 241

- "Triggering a Javadump"
- "Interpreting a Javadump" on page 242
- "Environment variables and Javadump" on page 262

Enabling a Javadump

Javadumps are enabled by default. You can turn off the production of Javadumps with **-Xdump:java:none**.

You are not recommended to turn off Javadumps because they are an essential diagnostic tool.

Use the **-Xdump:java** option to give more fine-grained control over the production of Javadumps. See "Using dump agents" on page 221 for more information.

Triggering a Javadump

Javadumps can be triggered by error conditions, or can be initiated in a number of ways to obtain diagnostic information.

Javadumps triggered by error conditions

By default, a Javadump is triggered when one of the following error conditions occurs:

An unrecoverable native exception
Not a Java Exception. An "unrecoverable" exception is one that causes the JVM to stop. The JVM handles the event by producing a system dump followed by a snap trace file, a Javadump, and then terminating the process.

The JVM has insufficient memory to continue operation
There are many reasons for running out of memory. See "Problem determination" on page 173 for more information.

Javadumps triggered by request

You can initiate a Javadump to obtain diagnostic information in one of the following ways:

You can send a signal to the JVM from the command line

The signal for z/OS is `SIGQUIT`. Use the command `kill -QUIT n` to send the signal to a process with process ID (PID) `n`. Alternatively, press **CTRL+V** in the shell window that started Java.

The JVM continues after the signal has been handled.

You can use the JavaDump() method in your application

The com.ibm.jvm.Dump class contains a static JavaDump() method that causes Java code to initiate a Javadump. In your application code, add a call to com.ibm.jvm.Dump.JavaDump(). This call is subject to the same Javadump environment variables that are described in "Enabling a Javadump."

The JVM continues after the Javadump is produced.

You can initiate a Javadump using the wasadmin utility

In a WebSphere Application Server environment, use the **wasadmin** utility to initiate a dump.

The JVM continues after the Javadump is produced.

You can configure a dump agent to trigger a Javadump
Use the **-Xdump:java:** option to configure a dump agent on the command line. See "Using the -Xdump option" on page 221 for more information.

You can use the trigger trace option to generate a Javadump
Use the **-Xtrace:trigger** option to produce a Javadump by calling the substring method shown in the following example:

```
-Xtrace:trigger=method{java/lang/String.substring,javadump}
```

For a detailed description of this trace option, see "trigger=<*clause*>[,<*clause*>][,<*clause*>]..." on page 308

Interpreting a Javadump

This section gives examples of the information contained in a Javadump and how it can be useful in problem solving.

The content and range of information in a Javadump might change between JVM versions or service refreshes. Some information might be missing, depending on the operating system platform and the nature of the event that produced the Javadump.

Javadump tags:

The Javadump file contains sections separated by eyecatcher title areas to aid readability of the Javadump.

The first such eyecatcher is shown as follows:

```
NULL           ------------------------------------------------------------
0SECTION       ENVINFO subcomponent dump routine
NULL           ================================
```

Different sections contain different tags, which make the file easier to parse for performing simple analysis.

You can also use DTFJ to parse a Javadump, see "Using the Diagnostic Tool Framework for Java" on page 405 for more information.

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

```
1CIJAVAVERSION JRE 1.7.0 z/OS s390-31 build 20110511_082084
(pmz3170-20110513_04)
```

Normal tags have these characteristics:

- Tags are up to 15 characters long (padded with spaces).
- The first digit is a nesting level (0,1,2,3). Nesting levels might be omitted, for example a level 2 tag might be followed by a level 4 tag.
- The second and third characters identify the section of the dump. The major sections are:
  CI Command-line interpreter
  CL Class loader
  LK Locking
  ST Storage (Memory management)
  TI Title
  XE Execution engine
- The remainder is a unique string, `JAVAVERSION` in the previous example.

Special tags have these characteristics:

- A tag of NULL means the line is just to aid readability.
- Every section is headed by a tag of 0SECTION with the section title.

Here is an example of some tags taken from the start of a dump.

```
NULL           ------------------------------------------------------------------
0SECTION       TITLE subcomponent dump routine
NULL           ================================
1TICHARSET     IBM-1047
1TISIGINFO     Dump Event "user" (00004000) received
1TIDATETIME    Date: 2011/05/13 at 17:37:23
1TIFILENAME    Javacore filename: /u/user1/javacore.20110513.173723.66276.0001.txt
1TIREQFLAGS    Request Flags: 0x1 (exclusive)
1TIPREPSTATE   Prep State: 0x104 (exclusive_vm_access+)
NULL           ------------------------------------------------------------------
0SECTION       GPINFO subcomponent dump routine
NULL           ================================
2XHOSLEVEL     OS Level : z/OS 01.08.00
2XHCPUS        Processors -
3XHCPUARCH       Architecture : s390
3XHNUMCPUS       How Many : 3
3XHNUMASUP       NUMA is either not supported or has been disabled by user
```

TITLE, GPINFO, and ENVINFO sections:

At the start of a Javadump, the first three sections are the TITLE, GPINFO, and ENVINFO sections. They provide useful information about the cause of the dump.

The following example shows some output taken from a simple Java test program calling (using JNI) an external function that causes a “general protection fault” (GPF).

TITLE
Shows basic information about the event that caused the generation of the Javadump, the time it was taken, and its name.

```
|  0SECTION       TITLE subcomponent dump routine
|  NULL           ================================
|  1TICHARSET     IBM-1047
|  1TISIGINFO     Dump Event "gpf" (00002000) received
|  1TIDATETIME    Date:                 2011/11/30 at 12:51:51
|  1TIFILENAME    Javacore filename:    /team/test/mz64/javacore.20111130.125148.33558225.0002.txt
|  1TIREQFLAGS    Request Flags: 0x1 (exclusive)
|  1TIPREPSTATE   Prep State: 0x100 (trace_disabled)
|  1TIPREPINFO    Exclusive VM access not taken: data may not be consistent across javacore sections
```

GPINFO
Varies in content depending on whether the Javadump was produced because of a GPF or not. It shows some general information about the operating system. The registers specific to the processor and architecture are also displayed.

```
|  0SECTION       GPINFO subcomponent dump routine
|  NULL           ================================
|  2XHOSLEVEL     OS Level         : z/OS 01.11.00
|  2XHCPUS        Processors -
|  3XHCPUARCH       Architecture   : s390x
|  3XHNUMCPUS       How Many       : 6
|  3XHNUMASUP       NUMA is either not supported or has been disabled by user
|  NULL
|  1XHEXCPCODE    J9Generic_Signal_Number: 00000004
|  1XHEXCPCODE    Signal_Number: 0000000B
|  1XHEXCPCODE    Error_Value: 00000000
|  1XHEXCPCODE    Signal_Code: 00000035
```

```
1XHEXCPCODE    Handler1: 0000000080871E300
1XHEXCPCODE    Handler2: 0000000808622D00
NULL
1XHREGISTERS   Registers:
2XHREGISTER      gpr0: 0000000000000000
2XHREGISTER      gpr1: 0000000839D10400
2XHREGISTER      gpr2: 0000000000000020
2XHREGISTER      gpr3: 000000086B9C0498
2XHREGISTER      gpr4: 00000008109FDDC0
2XHREGISTER      gpr5: 0000000808614A30
2XHREGISTER      gpr6: 00000000259A3710
2XHREGISTER      gpr7: 000000002500BEBA
2XHREGISTER      gpr8: 0000000000000007
....
1XHFLAGS       VM flags:0000000000000000
```

The GPINFO section also refers to the vmState, recorded in the console output as `VM flags`. The vmState is the thread-specific state of what was happening in the JVM at the time of the crash. The value for vmState is a hexadecimal number ending in MSSSS, where M is the SDK component and SSSS is component specific code.

SDK component	Code number
NONE	0x00000
INTERPRETER	0x10000
GC	0x20000
GROW_STACK	0x30000
JNI	0x40000
JIT_CODEGEN	0x50000
BCVERIFY	0x60000
RTVERIFY	0x70000
SHAREDCLASSES	0x80000

In the example, the value for vmState is `VM flags:0000000000000000`, which indicates a crash in code outside the SDK.

The crash was in the application native function `Java_Crash_segv`, as shown in the backtrace, which is in the THREADS section of the javacore:

```
0SECTION       THREADS subcomponent dump routine
NULL           ================================
....
1XMCURTHDINFO  Current thread
NULL           --------------
3XMTHREADINFO      "main" J9VMThread:0x0000000839D10400, j9thread_t:0x0000000808629CA0, java/lang/Thread:0x0000000819275148,
state:R, prio=5
3XMTHREADINFO1            (native thread ID:0x25562800, native priority:0x5, native policy:UNKNOWN)
3XMTHREADINFO3           Java callstack:
4XESTACKTRACE                at Crash.segv(Native Method)
4XESTACKTRACE                at Crash.main(Crash.java:27)
3XMTHREADINFO3           Native callstack:
....                         ....
4XENATIVESTACK               masterSynchSignalHandler+0xda52e6a8 (, 0x0000000000000000)
4XENATIVESTACK               __zerros+0xda9d5398 (, 0x0000000000000000)
4XENATIVESTACK               CEEHDSP+0xdae0c040 (, 0x0000000000000000)
4XENATIVESTACK               CEEOSIGJ+0xdabc4660 (, 0x0000000000000000)
4XENATIVESTACK               CELQHROD+0xdadf92f8 (, 0x0000000000000000)
4XENATIVESTACK               CEEOSIGG+0xdabcafb0 (, 0x0000000000000000)
4XENATIVESTACK               CELQHROD+0xdadf92f8 (, 0x0000000000000000)
4XENATIVESTACK               Java_Crash_segv+0xdaff4198 (, 0x0000000000000000)
4XENATIVESTACK               RUNCALLINMETHOD+0xda699f58 (, 0x0000000000000000)
4XENATIVESTACK               gpProtectedRunCallInMethod+0xda65db20 (, 0x0000000000000000)
4XENATIVESTACK               signalProtectAndRunGlue+0xda65dd40 (, 0x0000000000000000)
4XENATIVESTACK               j9sig_protect+0xda52cda0 (, 0x0000000000000000)
```

```
4XENATIVESTACK               gpCheckCallin+0xda65b360 (, 0x0000000000000000)
4XENATIVESTACK               callStaticVoidMethod+0xda664178 (, 0x0000000000000000)
4XENATIVESTACK               JavaMain+0xda78ee90 (, 0x0000000000000000)
```

When the vmState major component is JIT_CODEGEN, see the information at "JIT and AOT problem determination" on page 322.

ENVINFO

Shows information about the JRE level that failed and details about the command line that launched the JVM process and the JVM environment in place.

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.

```
0SECTION       ENVINFO subcomponent dump routine
NULL           ================================
1CIJAVAVERSION JRE 1.7.0 z/OS s390x-64 build 20111126_95926 (pmz6470sr1-20111128_01 (SR1))
1CIVMVERSION   VM build R26_JVM_26_20111125_1442_B95877
1CIJITVERSION  r11_20111122_21502
1CIGCVERSION   GC - R26_JVM_26_20111123_1546_B95682
1CIJITMODES    JIT enabled, AOT enabled, FSD disabled, HCR disabled
1CIRUNNINGAS   Running as a standalone JVM
1CIPROCESSID   Process ID: 14632 (0x3928)
1CICMDLINE     sdk/jre/bin/java Crash
1CIJAVAHOMEDIR Java Home Dir:   /team/test/mz64/sdk/jre
1CIJAVADLLDIR  Java DLL Dir:    /team/test/mz64/sdk/jre/bin
1CISYSCP       Sys Classpath:   /team/test/mz64/sdk/jre/lib/s390x/default/jclSC170/vm.jar...
1CIUSERARGS    UserArgs:
2CIUSERARG              -Xoptionsfile=/team/test/mz64/sdk/jre/lib/s390x/default/options.default
2CIUSERARG              -Xlockword:mode=default,noLockword=java/lang/String,noLockword=...
2CIUSERARG              -Xjcl:jclse7b_26
2CIUSERARG              -Dcom.ibm.oti.vm.bootstrap.library.path=/team/test/mz64/sdk/jre/lib/...
2CIUSERARG              -Dsun.boot.library.path=/team/test/mz64/sdk/jre/lib/s390x/default:...
2CIUSERARG              -Djava.library.path=/team/test/mz64/sdk/jre/lib/s390x/default:/team/...
2CIUSERARG              -Djava.home=/team/test/mz64/sdk/jre
2CIUSERARG              -Djava.ext.dirs=/team/test/mz64/sdk/jre/lib/ext
2CIUSERARG              -Duser.dir=/team/test/mz64....
```

The ENVINFO section of the javacore contains additional information about the operating system environment in which the JVM is running. This information includes:

- The system ulimits, or user limits, in place. These values are shown only on UNIX platforms.
- The system environment variables that are in force.

The output is similar to the following lines:

```
1CIUSERLIMITS  User Limits (in bytes except for NOFILE and NPROC)
NULL           ------------------------------------------------------------------------
NULL           type                            soft limit           hard limit
2CIUSERLIMIT   RLIMIT_AS                        unlimited            unlimited
2CIUSERLIMIT   RLIMIT_CORE                        4194304              4194304
2CIUSERLIMIT   RLIMIT_CPU                       unlimited            unlimited
2CIUSERLIMIT   RLIMIT_DATA                      unlimited            unlimited
2CIUSERLIMIT   RLIMIT_FSIZE                 8796093018112        8796093018112
2CIUSERLIMIT   RLIMIT_NOFILE                        10012                10012
2CIUSERLIMIT   RLIMIT_STACK                     unlimited            unlimited
2CIUSERLIMIT   RLIMIT_MEMLIMIT                21474836480          21474836480
NULL
1CIENVVARS     Environment Variables
NULL           ------------------------------------------------------------------------
2CIENVVAR      HOSTTYPE=i370
2CIENVVAR      EPHBookReadConfig=/etc/booksrv/bookread.conf
2CIENVVAR      LIBPATH=/team/test/mz64/sdk/jre/lib/s390x/default:/team/test/....
2CIENVVAR      _CC_CVERSION=0x41090000
```

```
| 2CIENVVAR      UNZIP=-:
| 2CIENVVAR      SHLVL=1
| 2CIENVVAR      HOSTNAME=TOROLABV
| 2CIENVVAR      SSH_TTY=/dev/ttyp0001
| 2CIENVVAR      JAVA_DUMP_TDUMP_PATTERN=J9BUILD.JVM.TDUMP.D%y%m%d.T%H%M%S
| 2CIENVVAR      _CC_PNAME=EDCPRLK
| 2CIENVVAR      _CC_CLIB_PREFIX=CBC
```

Native memory (NATIVEMEMINFO):

The NATIVEMEMINFO section of a Javadump provides information about the native memory allocated by the Java Runtime Environment (JRE).

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

When the JRE 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 JRE component. Each memory category contains the total value for each counter in that category and all related subcategories.

The JRE tracks native memory allocated only by the Java runtime environment and class libraries. The JRE 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           ------------------------------------------------------------------------
```

You can obtain additional diagnostic information about memory allocation for class library native code. The following output shows the extra information recorded in the `Class Libraries` section when the system property **-Dcom.ibm.dbgmalloc=true** is set:

```
|                 3MEMUSER       |  +--Standard Class Libraries: 17,816 bytes / 17 allocations
|                 3MEMUSER       |  |  |
|                 4MEMUSER       |  |  +--IO, Math and Language: 1,048 bytes / 1 allocation
|                 3MEMUSER       |  |  |
|                 4MEMUSER       |  |  +--Zip: 1,048 bytes / 1 allocation
|                 3MEMUSER       |  |  |
|                 4MEMUSER       |  |  +--Wrappers: 3,144 bytes / 3 allocations
|                 4MEMUSER       |  |  |  |
|                 5MEMUSER       |  |  |  +--Malloc: 1,048 bytes / 1 allocation
|                 4MEMUSER       |  |  |  |
|                 5MEMUSER       |  |  |  +--z/OS EBCDIC Conversion: 1,048 bytes / 1 allocation
|                 4MEMUSER       |  |  |  |
|                 5MEMUSER       |  |  |  +--Other: 1,048 bytes / 1 allocation
|                 3MEMUSER       |  |  |
|                 4MEMUSER       |  |  +--Networking: 4,192 bytes / 4 allocations
|                 4MEMUSER       |  |  |  |
|                 5MEMUSER       |  |  |  +--NET: 1,048 bytes / 1 allocation
|                 4MEMUSER       |  |  |  |
|                 5MEMUSER       |  |  |  +--NIO and NIO.2: 1,048 bytes / 1 allocation
|                 4MEMUSER       |  |  |  |
|                 5MEMUSER       |  |  |  +--RMI: 1,048 bytes / 1 allocation
|                 4MEMUSER       |  |  |  |
|                 5MEMUSER       |  |  |  +--Other: 1,048 bytes / 1 allocation
|                 3MEMUSER       |  |  |
|                 4MEMUSER       |  |  +--GUI: 5,240 bytes / 5 allocations
|                 4MEMUSER       |  |  |  |
|                 5MEMUSER       |  |  |  +--AWT: 1,048 bytes / 1 allocation
```

```
4MEMUSER       |  |  |  |
5MEMUSER       |  |  |  +--MAWT: 1,048 bytes / 1 allocation
4MEMUSER       |  |  |  |
5MEMUSER       |  |  |  +--JAWT: 1,048 bytes / 1 allocation
4MEMUSER       |  |  |  |
5MEMUSER       |  |  |  +--Medialib Image: 1,048 bytes / 1 allocation
4MEMUSER       |  |  |  |
5MEMUSER       |  |  |  +--Other: 1,048 bytes / 1 allocation
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Font: 1,048 bytes / 1 allocation
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Sound: 1,048 bytes / 1 allocation
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Other: 1,048 bytes / 1 allocation
```

For more information about using **-Dcom.ibm.dbgmalloc=true**, see "System property command-line options" on page 419.

Storage Management (MEMINFO):

The MEMINFO section provides information about the Memory Manager.

The MEMINFO section, giving information about the Memory Manager, follows the first three sections. See "Memory management" on page 23 for details about how the Memory Manager works.

This part of the Javadump provides various storage management values in hexadecimal. The information also shows the free memory, used memory and total memory for the heap, in decimal and hexadecimal. 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 "Garbage Collector command-line options" on page 453.

This section also contains garbage collection history data, described in "Default memory management tracing" on page 290. Garbage collection history data is shown as a sequence of tracepoints, each with a timestamp, ordered with the most recent tracepoint first.

In the Javadump, segments are blocks of memory allocated by the Java runtime environment for tasks that use large amounts of memory. Example tasks are maintaining JIT caches, and storing Java classes. The Java runtime environment also allocates other native memory, that is not listed in the MEMINFO section. The total memory used by Java runtime segments does not necessarily represent the complete memory footprint of the Java runtime environment. A Java runtime segment consist of the segment data structure, and an associated block of native memory.

The following example shows some typical output. All the values are output 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   0x0000000000042D4B0         --                --                --         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 (0x0000000000200A9C)
1STSEGINUSE    Total memory in use:    1959236 (0x000000000001DE544)
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 (0x000000000001FE108)
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.

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 257.

```
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:CW,
| prio=5
| 3XMJAVALTHREAD (java/lang/Thread getId:0x1, isDaemon:false)
| 3XMTHREADINFO1 (native thread ID:0xE28, native priority:0x5, 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=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)
```

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 properties on the next line are the native operating system thread ID, native operating system thread priority and native operating system scheduling policy.

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 257.

For Java threads and attached native threads, the output might contain 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 presence and contents of this line depends on your operating system. If your operating system does not support the reporting of CPU times for each thread, this line is absent. Some operating systems report only the total CPU time that is consumed by a thread, and others also report the time that is consumed in user code and in system code.

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.

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 are in CW (Condition Wait) state.

```
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)
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)
```

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" (0x41481900)
```

Understanding Java and native thread details:

Below each thread heading are the stack traces, which can be separated into three types; Java threads, attached native threads and unattached native threads.

By default, Javadumps contain native stack traces for all threads on AIX, Linux, and 32-bit Windows. Each native thread is paired with the corresponding Java thread, if one exists. On AIX and Linux platforms, the JVM delivers a SIGRTMIN control signal to each native thread in response to a request for a Javadump. You can disable this feature by controlling the dump agent. See the **preempt** option, detailed in the "request option" on page 233 topic. Native stack traces are not available on 64-bit Windows, 31-bit z/OS and 64-bit z/OS.

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" on page 257. 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)
```

Attached native thread

The attached native threads provide the same set of information as a Java and native thread pair, but do not have a Java stack trace. For example:

```
"JIT Compilation Thread" TID:0x01E92300, j9thread_t:0x00295780, state:CW, prio=10
            (native thread ID:0x3030, native priority:0xB, native policy:UNKNOWN)
          No Java callstack associated with this thread
          Native callstack:
              KiFastSystemCallRet+0x0 (0x7C82860C [ntdll+0x2860c])
              WaitForSingleObject+0x12 (0x77E61C8D [kernel32+0x21c8d])
              monitor_wait_original+0x5a0 (j9thread.c:3593, 0x7FFA49F0 [J9THR26+0x49f0])
              monitor_wait+0x5f (j9thread.c:3439, 0x7FFA443F [J9THR26+0x443f])
              j9thread_monitor_wait+0x14 (j9thread.c:3309, 0x7FFA4354 [J9THR26+0x4354])
              TR_J9Monitor::wait+0x13 (monitor.cpp:61, 0x009B5403 [j9jit26+0x585403])
              protectedCompilationThreadProc+0x2a4 (compilationthread.cpp:2063, 0x0043A284
[j9jit26+0xa284])
              j9sig_protect+0x42 (j9signal.c:144, 0x7FE117B2 [J9PRT26+0x117b2])
              compilationThreadProc+0x123 (compilationthread.cpp:1996, 0x00439F93 [j9jit26+0x9f93])
              thread_wrapper+0x133 (j9thread.c:975, 0x7FFA1FE3 [J9THR26+0x1fe3])
              _threadstart+0x6c (thread.c:196, 0x7C34940F [msvcr71+0x940f])
              GetModuleHandleA+0xdf (0x77E6482F [kernel32+0x2482f])
```

Unattached native thread

The unattached native threads do not have meaningful names and provide only minimal information in addition to the stack trace, for example:

```
  Anonymous native thread
            (native thread ID:0x229C, native priority: 0x0, native policy:UNKNOWN)
          Native callstack:
              KiFastSystemCallRet+0x0 (0x7C82860C [ntdll+0x2860c])
              WaitForSingleObject+0x12 (0x77E61C8D [kernel32+0x21c8d])
              j9thread_sleep_interruptable+0x1a7 (j9thread.c:1475, 0x7FFA24C7 [J9THR26+0x24c7])
              samplerThreadProc+0x261 (hookedbythejit.cpp:3227, 0x00449C21 [j9jit26+0x19c21])
```

```
thread_wrapper+0x133 (j9thread.c:975, 0x7FFA1FE3 [J9THR26+0x1fe3])
_threadstart+0x6c (thread.c:196, 0x7C34940F [msvcr71+0x940f])
GetModuleHandleA+0xdf (0x77E6482F [kernel32+0x2482f])
```

Java dumps are triggered in two distinct ways that influence the structure of the `THREADS` section:

A general protection fault (GPF) occurs:
The `Current thread` subsection contains only the thread that generated the GPF. The other threads are shown in the `Thread Details` subsection.

A user requests a Java dump for an event using, for example, the kill -QUIT command or the com.ibm.jvm.Dump.JavaDump API:
There is no `Current thread` subsection and all threads are shown in the `Thread Details` subsection.

The following example is an extract from the `THREADS` section that was generated when the `main` thread caused a GPF.

```
NULL           ------------------------------------------------------------------------
0SECTION       THREADS subcomponent dump routine
NULL           ================================
NULL
1XMCURTHDINFO  Current thread
NULL           ----------------------
3XMTHREADINFO      "main" TID:0x01E91E00, j9thread_t:0x00295518, state:R, prio=5
3XMTHREADINFO1            (native thread ID:0x3C34, native priority:0x5, native policy:UNKNOWN)
3XMTHREADINFO3           No Java callstack associated with this thread
3XMTHREADINFO3           Native callstack:
4XENATIVESTACK               doTriggerActionSegv+0xe (trigger.c:1880, 0x7FC7930E [j9trc26+0x930e])
4XENATIVESTACK               triggerHit+0x7d (trigger.c:2427, 0x7FC79CAD [j9trc26+0x9cad])
4XENATIVESTACK               twTriggerHit+0x24 (tracewrappers.c:123, 0x7FC71394 [j9trc26+0x1394])
4XENATIVESTACK               utsTraceV+0x14c (ut_trace.c:2105, 0x7FB64F1C [j9ute26+0x14f1c])
4XENATIVESTACK               j9Trace+0x5a (tracewrappers.c:732, 0x7FC724DA [j9trc26+0x24da])
4XENATIVESTACK               jvmtiHookVMShutdownLast+0x33 (jvmtihook.c:1172, 0x7FBA5C63
[j9jvmti26+0x15c63])
4XENATIVESTACK               J9HookDispatch+0xcf (hookable.c:175, 0x7FD211CF [J9HOOKABLE26+0x11cf])
4XENATIVESTACK               protectedDestroyJavaVM+0x224 (jniinv.c:323, 0x7FE50D84 [j9vm26+0x20d84])
4XENATIVESTACK               j9sig_protect+0x42 (j9signal.c:144, 0x7FE117B2 [J9PRT26+0x117b2])
4XENATIVESTACK               DestroyJavaVM+0x206 (jniinv.c:482, 0x7FE50B06 [j9vm26+0x20b06])
4XENATIVESTACK               DestroyJavaVM+0xe (jvm.c:332, 0x7FA1248E [jvm+0x248e])
4XENATIVESTACK               newStringCp1252+0x22 (jni_util.c:511, 0x00403769 [java+0x3769])
4XENATIVESTACK               wcp+0x48 (canonicalize_md.c:78, 0x00409615 [java+0x9615])
4XENATIVESTACK               GetModuleHandleA+0xdf (0x77E6482F [kernel32+0x2482f])
NULL
NULL
1XMTHDINFO     Thread Details
NULL           -----------------
NULL
3XMTHREADINFO      Anonymous native thread
3XMTHREADINFO1            (native thread ID:0x175C, native priority: 0x0, native policy:UNKNOWN)
3XMTHREADINFO3           Native callstack:
4XENATIVESTACK               KiFastSystemCallRet+0x0 (0x7C82860C [ntdll+0x2860c])
4XENATIVESTACK               WaitForSingleObject+0x12 (0x77E61C8D [kernel32+0x21c8d])
4XENATIVESTACK               JNU_GetStringPlatformChars+0x2 (jni_util.c:795, 0x00404683 [java+0x4683])
4XENATIVESTACK               JNU_ClassObject+0x10 (jni_util.c:897, 0x00403B06 [java+0x3b06])
NULL
3XMTHREADINFO      "JIT Compilation Thread" TID:0x01E92300, j9thread_t:0x00295780, state:CW, prio=10
3XMTHREADINFO1            (native thread ID:0x3030, native priority:0xB, native policy:UNKNOWN)
3XMTHREADINFO3           No Java callstack associated with this thread
3XMTHREADINFO3           Native callstack:
4XENATIVESTACK               KiFastSystemCallRet+0x0 (0x7C82860C [ntdll+0x2860c])
4XENATIVESTACK               WaitForSingleObject+0x12 (0x77E61C8D [kernel32+0x21c8d])
4XENATIVESTACK               monitor_wait_original+0x5a0 (j9thread.c:3593, 0x7FFA49F0 [J9THR26+0x49f0])
4XENATIVESTACK               monitor_wait+0x5f (j9thread.c:3439, 0x7FFA443F [J9THR26+0x443f])
4XENATIVESTACK               j9thread_monitor_wait+0x14 (j9thread.c:3309, 0x7FFA4354 [J9THR26+0x4354])
```

```
4XENATIVESTACK               TR_J9Monitor::wait+0x13 (monitor.cpp:61, 0x009B5403 [j9jit26+0x585403])
4XENATIVESTACK               protectedCompilationThreadProc+0x2a4 (compilationthread.cpp:2063,
0x0043A284 [j9jit26+0xa284])
4XENATIVESTACK               j9sig_protect+0x42 (j9signal.c:144, 0x7FE117B2 [J9PRT26+0x117b2])
NULL
3XMTHREADINFO      Anonymous native thread
3XMTHREADINFO1            (native thread ID:0x229C, native priority: 0x0, native policy:UNKNOWN)
3XMTHREADINFO3           Native callstack:
4XENATIVESTACK               KiFastSystemCallRet+0x0 (0x7C82860C [ntdll+0x2860c])
4XENATIVESTACK               WaitForSingleObject+0x12 (0x77E61C8D [kernel32+0x21c8d])
4XENATIVESTACK               j9thread_sleep_interruptable+0x1a7 (j9thread.c:1475, 0x7FFA24C7
[J9THR26+0x24c7])
4XENATIVESTACK               samplerThreadProc+0x261 (hookedbythejit.cpp:3227, 0x00449C21
[j9jit26+0x19c21])
4XENATIVESTACK               thread_wrapper+0x133 (j9thread.c:975, 0x7FFA1FE3 [J9THR26+0x1fe3])
4XENATIVESTACK               _threadstart+0x6c (thread.c:196, 0x7C34940F [msvcr71+0x940f])
4XENATIVESTACK               GetModuleHandleA+0xdf (0x77E6482F [kernel32+0x2482f])
NULL
```

On Linux, there are a number of frames reported for threads that are part of the backtrace mechanism. To find the point in the backtrace at which the GPF occurred, look for the frame that has no associated file and offset information. In the following example, this frame is (0xFFFFE600).

```
1XMCURTHDINFO  Current thread
NULL           ----------------------
3XMTHREADINFO      "(unnamed thread)" J9VMThread:0x0806C500, j9thread_t:0x0804F420, java/lang/
Thread:0x00000000, state:R, prio=0
3XMTHREADINFO1            (native thread ID:0x7710, native priority:0x5, native policy:UNKNOWN)
3XMTHREADINFO2            (native stack address range from:0xF75B7000, to:0xF7FB8000, size:0xA01000)
3XMTHREADINFO3           No Java callstack associated with this thread
3XMTHREADINFO3           Native callstack:
4XENATIVESTACK               (0xF74FA32F [libj9prt26.so+0x0])
4XENATIVESTACK               (0xF7508B6B [libj9prt26.so+0x0])
4XENATIVESTACK               (0xF74FA02E [libj9prt26.so+0x0])
4XENATIVESTACK               (0xF74FA123 [libj9prt26.so+0x0])
[....]
4XENATIVESTACK               (0xF7533507 [libj9vm26.so+0x0])
4XENATIVESTACK               (0xF7508B6B [libj9prt26.so+0x0])
4XENATIVESTACK               (0xF753317C [libj9vm26.so+0x0])
4XENATIVESTACK               (0xF75086E7 [libj9prt26.so+0x0])
4XENATIVESTACK               (0xFFFFE600)
4XENATIVESTACK               (0xF7216F2D [libj9trc26.so+0x0])
4XENATIVESTACK               (0xF7216278 [libj9trc26.so+0x0])
4XENATIVESTACK               (0xF7FD8C7E [libj9hookable26.so+0x0])
4XENATIVESTACK               (0xF75391C1 [libj9vm26.so+0x0])
[....]
```

The frame descriptions in the call stacks have the following format. Items that are unavailable can be omitted, except for the instruction pointer.

```
SYMBOL+SYMBOL_OFFSET (ID, INSTRUCTION_POINTER [MODULE+MODULE_OFFSET])
```

The regular expression pattern is:

```
 (?:([\S^+]+?)(?:\+(0x(?:[0-9A-Fa-f])+))? )?\((?:([^,]+), )?(0x(?:[0-9A-Fa-f])+)
(?: \[([\S^+]+?)(?:\+(0x(?:[0-9A-Fa-f])+))\])?\)
```

The group IDs are:

```
SYMBOL = 1
SYMBOL_OFFSET = 2
ID = 3
IP = 4
MODULE = 5
MODULE_OFFSET = 6
```

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.

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 362 for a description of the summary information.

```
------------------------------------------------------------------------
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
```

Class loaders and Classes (CLASSES):

An example of the classloader (CLASSES) section that includes Classloader summaries and Classloader loaded classes. Classloader summaries are the defined class loaders and the relationship between them. Classloader loaded classes are the classes that are loaded by each class loader.

See "Class loading" on page 53 for information about the parent-delegation model.

In this example, there are the standard three class loaders:

- Application class loader (sun/misc/Launcher$AppClassLoader), which is a child of the extension class loader.
- The Extension class loader (sun/misc/Launcher$ExtClassLoader), which is a child of the bootstrap class loader.
- The Bootstrap class loader. Also known as the System class loader.

The example that follows shows this relationship. Take the application class loader with the full name sun/misc/Launcher$AppClassLoader. Under `Classloader summaries`, it has flags `-----ta-`, which show that the class loader is t=trusted and a=application (See the example for information on class loader flags). It gives the number of loaded classes (1) and the parent class loader as sun/misc/Launcher$ExtClassLoader.

Under the `ClassLoader loaded classes` heading, you can see that the application class loader has loaded three classes, one called Test at address 0x41E6CFE0.

In this example, the System class loader has loaded a large number of classes, which provide the basic set from which all applications derive.

```
------------------------------------------------------------------------
CLASSES subcomponent dump routine
================================
Classloader summaries
        12345678: 1=primordial,2=extension,3=shareable,4=middleware,
                  5=system,6=trusted,7=application,8=delegating
        p---st--        Loader *System*(0x00439130)
                        Number of loaded libraries 5
                        Number of loaded classes 306
                        Number of shared classes 306
        -x--st--        Loader sun/misc/Launcher$ExtClassLoader(0x004799E8),
                        Parent *none*(0x00000000)
                        Number of loaded classes 0
        -----ta-        Loader sun/misc/Launcher$AppClassLoader(0x00484AD8),
                        Parent sun/misc/Launcher$ExtClassLoader(0x004799E8)
                        Number of loaded classes 1
ClassLoader loaded classes
Loader *System*(0x00439130)
  java/security/CodeSource(0x41DA00A8)
  java/security/PermissionCollection(0x41DA0690)
    << 301 classes removed for clarity >>
  java/util/AbstractMap(0x4155A8C0)
  java/io/OutputStream(0x4155ACB8)
  java/io/FilterOutputStream(0x4155AE70)
Loader sun/misc/Launcher$ExtClassLoader(0x004799E8)
Loader sun/misc/Launcher$AppClassLoader(0x00484AD8)
  Test(0x41E6CFE0)
  Test$DeadlockThread0(0x41E6D410)
  Test$DeadlockThread1(0x41E6D6E0)
```

Environment variables and Javadump

Although the preferred mechanism of controlling the production of Javadumps is now by the use of dump agents using **-Xdump:java**, you can also use the previous mechanism, environment variables.

The following table details environment variables specifically concerned with Javadump production:

Environment Variable	Usage Information
DISABLE_JAVADUMP	Setting **DISABLE_JAVADUMP** to true is the equivalent of using **–Xdump:java:none** and stops the default production of javadumps.
IBM_JAVACOREDIR	The default location into which the Javacore will be written.
JAVA_DUMP_OPTS	Use this environment variable to control the conditions under which Javadumps (and other dumps) are produced. See "Dump agent environment variables" on page 237 for more information.
IBM_JAVADUMP_OUTOFMEMORY	By setting this environment variable to false, you disable Javadumps for an out-of-memory exception.

Using Heapdump

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

There are two Heapdump formats, the text or classic Heapdump format and the Portable Heap Dump (PHD) format. Although the text or classic format is human-readable, the PHD format is compressed and is not human-readable. Both Heapdump formats contain a list of all object instances in the heap, including each object address, type or class name, size, and references to other objects. The Heapdumps also contain information about the version of the JVM that produced the Heapdump. They do not contain any object content or data other than the class names and the values (addresses) of the references.

You can use various tools on the Heapdump output to analyze the composition of the objects on the heap. This analysis might help to find the objects that are controlling large amounts of memory on the Java heap and determine why the Garbage Collector cannot collect them.

This chapter describes:

Getting Heapdumps

By default, a Heapdump is produced when the Java heap is exhausted. Heapdumps can be generated in other situations by use of **-Xdump:heap**.

To see which events will trigger a dump, use **-Xdump:what**. See "Using dump agents" on page 221 for more information about the **-Xdump** parameter.

You can also use the com.ibm.jvm.Dump.HeapDump() method in your application code, to generate a Heapdump programmatically.

By default, Heapdumps are produced in PHD format. To produce Heapdumps in text format, see "Enabling text formatted ("classic") Heapdumps."

Environment variables can also affect the generation of Heapdumps (although this is a deprecated mechanism). See "Environment variables and Heapdump" on page 264 for more details.

Enabling text formatted ("classic") Heapdumps:

The generated Heapdump is by default in the binary, platform-independent, PHD format, which can be examined using the available tooling.

For more information, see "Available tools for processing Heapdumps." However, an immediately readable view of the heap is sometimes useful. You can obtain this view by using the **opts=** suboption with **-Xdump:heap** (see "Using dump agents" on page 221). For example:

- **-Xdump:heap:opts=CLASSIC** will start the default Heapdump agents using classic rather than PHD output.
- **-Xdump:heap:defaults:opts=CLASSIC+PHD** will enable both classic and PHD output by default for all Heapdump agents.

You can also define one of the following environment variables:

- **IBM_JAVA_HEAPDUMP_TEST**, which allows you to perform the equivalent of **opts=PHD+CLASSIC**
- **IBM_JAVA_HEAPDUMP_TEXT**, which allows the equivalent of **opts=CLASSIC**

Available tools for processing Heapdumps

There are several tools available for Heapdump analysis through IBM support Web sites.

The preferred Heapdump analysis tool is the IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer. The tool is available in IBM Support Assistant: http://www.ibm.com/software/support/isa/. Information about the tool can be found at http://www.ibm.com/developerworks/java/jdk/tools/memoryanalyzer/

Further details of the range of available tools can be found at http://www.ibm.com/support/docview.wss?uid=swg24009436

Using -Xverbose:gc to obtain heap information

Use the **-Xverbose:gc** utility to obtain information about the Java Object heap in real time while running your Java applications.

To activate this utility, run Java with the **-verbose:gc** option:

```
java -verbose:gc
```

For more information, see "Memory management" on page 23.

Environment variables and Heapdump

Although the preferred mechanism for controlling the production of Heapdumps is now the use of dump agents with **-Xdump:heap**, you can also use the previous mechanism, environment variables.

The following table details environment variables specifically concerned with Heapdump production:

Environment Variable	Usage Information
IBM_HEAPDUMP **IBM_HEAP_DUMP**	Setting either of these to any value (such as true) enables heap dump production by means of signals.
IBM_HEAPDUMPDIR	The default location into which the Heapdump will be written.
JAVA_DUMP_OPTS	Use this environment variable to control the conditions under which Heapdumps (and other dumps) are produced. See "Dump agent environment variables" on page 237 for more information .
IBM_HEAPDUMP_OUTOFMEMORY	By setting this environment variable to false, you disable Heapdumps for an OutOfMemory condition.
IBM_JAVA_HEAPDUMP_TEST	Use this environment variable to cause the JVM to generate both phd and text versions of Heapdumps. Equivalent to **opts=PHD+CLASSIC** on the **-Xdump:heap** option.
IBM_JAVA_HEAPDUMP_TEXT	Use this environment variable to cause the JVM to generate a text (human readable) Heapdump. Equivalent to **opts=CLASSIC** on the **-Xdump:heap** option.

Text (classic) Heapdump file format

The text or classic Heapdump is a list of all object instances in the heap, including object type, size, and references between objects. On z/OS, the Heapdump is in EBCDIC.

Header record

The header record is a single record containing a string of version information.

```
// Version: <version string containing SDK level, platform and JVM build level>
```

For example:

```
// Version: JRE 1.7.0 z/OS s390x-64 build 20120817_119700 (pmz6470sr3-20120821_01(SR3))
```

Object records

Object records are multiple records, one for each object instance on the heap, providing object address, size, type, and references from the object.

```
<object address, in hexadecimal> [<length in bytes of object instance, in decimal>] OBJ <object type>
<heap reference, in hexadecimal> <heap reference, in hexadecimal> ...
```

The object type is either a class name, or a class array type, or a primitive array type, shown by the standard JVM type signature, see "Java VM type signatures" on page 266

on page 266. Package names are included in the class names. References found in the object instance are listed, excluding references to an object's class and excluding null references.

Examples:

An object instance, length 32 bytes, of type java/lang/String, with its single reference to a char array:

```
0x000007FFFFF84278 [32] OBJ java/lang/String
   0x000007FFFFF842F0
```

An object instance, length 72 bytes, of type char array, as referenced from the java/lang/String:

```
0x000007FFFFF842F0 [32] OBJ [C
```

An object instance, length 48 bytes, of type array of java/lang/String

```
0x000007FFFFF84CB8 [48] OBJ [Ljava/lang/String;
   0x000007FFFFF84D70 0x000007FFFFF84D90 0x000007FFFFF84DB0 0x000007FFFFF84DD0
```

Class records

Class records are multiple records, one for each loaded class, providing class block address, size, type, and references from the class.

```
<class object address, in hexadecimal> [<length in bytes of class object, in decimal>] CLS <class type>
<heap reference, in hexadecimal> <heap reference, in hexadecimal>...
```

The class type is either a class name, or a class array type, or a primitive array type, shown by its standard JVM type signature, see "Java VM type signatures" on page 266. Package names are included in the class names. References found in the class block are listed, excluding null references.

Examples:

A class object, length 80 bytes, for class java/util/Vector, with heap references:

```
0x000007FFDFFC2F80 [80] CLS java/util/Vector
   0x000007FFFFF30A90 0x000007FFDFFC3030
```

Trailer record 1

Trailer record 1 is a single record containing record counts.

```
// Breakdown - Classes: <class record count, in decimal>,
Objects: <object record count, in decimal>,
ObjectArrays: <object array record count, in decimal>,
PrimitiveArrays: <primitive array record count, in decimal>
```

For example:

```
// Breakdown - Classes: 321, Objects: 3718, ObjectArrays: 169,
PrimitiveArrays: 2141
```

Trailer record 2

Trailer record 2 is a single record containing totals.

```
// EOF:  Total 'Objects',Refs(null) :
<total object count, in decimal>,
<total reference count, in decimal>
(,total null reference count, in decimal>)
```

For example:

```
// EOF:  Total 'Objects',Refs(null) : 6349,23240(7282)
```

Java VM type signatures

The Java VM type signatures are abbreviations of the Java types are shown in the following table:

Java VM type signatures	Java type
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L <*fully qualified-class*> ;	<*fully qualified-class*>
[<*type*>	<*type*>[ ] (array of <*type*>)
(<*arg-types*>) <*ret-type*>	method

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 for Java 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 for Java 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.

Different versions of PHD are produced, depending on the version of the J9 virtual machine (VM):

- J9 VM versions 2.4, 2.5, and 2.6 produce PHD version 5.
- J9 VM version 2.3 produces PHD version 4, until IBM SDK for Java 5.0 service refresh 9. From service refresh 10, the J9 VM version 2.3 produces PHD version 5. See APAR IZ34218.

To find out which IBM J9 VM you are using with the IBM SDK or JRE, type **java -version** on the command line and inspect the output.

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 system dumps and the dump viewer

The JVM can generate native system dumps, also known as core dumps, under configurable conditions.

System dumps contain a binary copy of process memory and are not human-readable. The dumps contain a complete copy of the Java heap, including the contents of all Java objects in the application. If you require a dump that does not contain this application data, see these topics: "Using Javadump" on page 240 or "Using Heapdump" on page 262.

Dump agents are the preferred method for controlling the generation of system dumps. For more information, see "Using dump agents" on page 221. To maintain compatibility with earlier versions, the JVM supports the use of environment variables for triggering system dumps. For more information, see "Dump agent environment variables" on page 237.

The dump viewer that is provided in the SDK is a cross-platform line-mode tool for viewing and analyzing system dumps. The following operating system tools can also be used:

- AIX: **dbx**
- Linux: **gdb**
- Windows: **windbg**
- z/OS: **ISPF**

This chapter tells you about system dumps and how to use the dump viewer. It contains these topics:

- "Overview of system dumps"
- "System dump defaults"
- "Using the dump viewer" on page 273

Overview of system dumps

The JVM can produce system dumps in response to specific events. A system dump is a raw binary dump of the process memory when the dump agent is triggered by a failure or by an event for which a dump is requested.

Generally, you use a tool to examine the contents of a system dump. A dump viewer tool is provided in the SDK, as described in this section, or you could use a platform-specific debugger to examine the dump.

For dumps triggered by a General Protection Fault (GPF), dumps produced by the JVM contain some context information that you can read. You can find this failure context information by searching in the dump for the eye-catcher

```
J9Generic_Signal_Number
```

For example:

```
J9Generic_Signal_Number=00000004 ExceptionCode=c0000005 ExceptionAddress=7FAB506D ContextFlags=0001003f
Handler1=7FEF79C0 Handler2=7FED8CF0 InaccessibleAddress=0000001C
EDI=41FEC3F0 ESI=00000000 EAX=41FB0E60 EBX=41EE6C01
ECX=41C5F9C0 EDX=41FB0E60
EIP=7FAB506D ESP=41C5F948 EBP=41EE6CA4
Module=E:\testjava\java7-32\sdk\jre\bin\j9jit24.dll
Module_base_address=7F8D0000 Offset_in_DLL=001e506d

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

Dump agents are the primary method for controlling the generation of system dumps. See "Using dump agents" on page 221 for more information on dump agents.

System dump defaults

There are default agents for producing system dumps when using the JVM.

Using the **-Xdump:what** option shows the following system dump agent:

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

This output shows that by default a system dump is produced in these cases:

- A general protection fault occurs. (For example, branching to memory location 0, or a protection exception.)
- An abort is encountered. (For example, native code has called abort() or when using `kill -ABRT` on Linux)

Attention: The JVM used to produce this output when a SIGSEGV signal was encountered. This behavior is no longer supported. Use the ABRT signal to produce dumps.

Using the dump viewer

System dumps are produced in a platform-specific binary format, typically as a raw memory image of the process that was running at the time the dump was initiated. The SDK dump viewer allows you to navigate around the dump, and obtain information in a readable form, with symbolic (source code) data where possible.

You can view Java information (for example, threads and objects on the heap) and native information (for example, native stacks, libraries, and raw memory locations). You can run the dump viewer on one platform to work with dumps from another platform. For example, you can look at Linux dumps on a Windows platform.

You can also explore the dump file by using IBM Monitoring and Diagnostic Tools for Java - Interactive Diagnostic Data Explorer. This tool is a GUI-based version of the dump viewer, which provides extra functionality such as command assistance and the ability to save the tool output.

Dump viewer: `jdmpview`

The dump viewer is a command-line tool that allows you to examine the contents of system dumps produced from the JVM. The dump viewer requires metadata created by the **jextract** utility, if the system dump was generated by a version of the IBM J9 virtual machine before V2.6. To check the version of a JVM, use the `java -version` command and examine the output. The dump viewer allows you to view both Java and native information from the time the dump was produced.

jdmpview is in the directory `sdk/bin`.

To start **jdmpview**, from a shell prompt, enter:

```
jdmpview  -zip <zip file>
```

or

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

The **jdmpview** tool accepts these parameters:

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

-notemp
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.

-xml *<xml file>*
Specify a metadata file. **jdmpview** guesses the name of the XML file if the **-xml** option is not present. This option is not required for core files generated from an IBM J9 2.6 virtual machine.

-zip *<zip file>*
Specify a compressed file containing the core file and associated XML file (produced by **jextract**).

Note: The **-core** and **-xml** options can be used with the **-zip** option to specify the core and XML files in the compressed file. Without the **-core** or **-xml** options,

jdmpview shows multiple contexts, one for each source file that it identified in the compressed file. For more information, see "Support for compressed files."

On z/OS, you can copy the dump to an HFS file and supply that as input to **jdmpview**, or you can supply a fully qualified MVS data set name. For example:

```
> jdmpview -core USER1.JVM.TDUMP.SSHD6.D070430.T092211
Loading image from DTFJ...
DTFJView version 1.0.24
Using DTFJ API version 1.3
```

After **jdmpview** processes the arguments with which it was started, it shows this message:

```
For a list of commands, type "help"; for how to use "help", type "help help"
>
```

When you see this message, you can start using commands.

When **jdmpview** is used with the **-zip** option, temporary disk space is required to uncompress the dump files from the compressed file. **jdmpview** uses the system temporary directory, `/tmp` on AIX, Linux, or zOS. An alternative temporary directory can be specified using the Java system property **java.io.tmpdir**. **jdmpview** shows an error message if insufficient disk space is available in the temporary directory. Use the **-notemp** option to prevent **jdmpview** from creating these temporary files. The temporary files are deleted when **jdmpview** exits or when you enter the **close** command on the command line.

You can significantly improve the performance of **jdmpview** against large dumps by ensuring that your system has enough memory available to avoid paging. On large dumps (that is, ones with large numbers of objects on the heap), you might have to run **jdmpview** using the **-Xmx** option to increase the maximum heap available. You might also have to increase the maximum heap size if you use the **-notemp** option, especially if you are analyzing a large heap, because this option specifies that all analysis is done in memory.

```
jdmpview  -J-Xmx<n> -zip <zip file>
```

To pass command-line arguments to the JVM, you must prefix them with **-J**.

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.

Example 1

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

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

Source : file:/C:/test/core.zip#core.20120329.165054.4176.0001.dmp
         *0 : PID: 4176 : JRE 1.7.0 Windows 7 amd64-64 build 20120228_104045 (pwa6470sr1-20120302_01(SR1) )
```

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.SVCDUMP.D120228.T153207.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 1.7.0 z/OS s390x-64 build 20120228_104045 (pmz6470sr1-20120302_01(SR1))
          6 : ASID: 0x73 EDB: 0x83d2053a0 : JRE 1.7.0 z/OS s390x-64 build 20120228_104045 (pmz6470sr1-20120302_01(SR1))
          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, Java dump, and heap dump files:

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

Source : file:///D:/Samples/multi-image.zip#core1.dmp
             *0 : PID: 10463 : JRE 1.7.0 Linux amd64-64 build 20120228_104045 pxa6470sr1-20120302_01(SR1))

Source : file:///D:/Samples/multi-image.zip#core2.dmp
              1 : PID: 12268 : JRE 1.7.0 Linux amd64-64 build 20120228_104045 pxa6470sr1-20120302_01(SR1))

Source : file:///D:/Samples/multi-image.zip#javacore.20120228.100128.10441.0002.txt
              2 : JRE 1.7.0 Linux amd64-64 build 20120228_94967 (pxa6470sr1-20120228_01(SR1))

Source : file:///D:/Samples/multi-image.zip#javacore.20120228.090916.14653.0002.txt
              3 : JRE 1.7.0 Linux amd64-64 build 20120228_94967 (pxa6470sr1-20120228_01(SR1))

Source : file:///D:/Samples/multi-image.zip#heapdump.20130711.093819.4336.0001.phd
              4 : JRE 1.7.0 Windows 7 amd64-64 build 20130711_156087 (pwa6470sr1-20111107_01(SR1))
```

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.20130711.093819.4336.0001.phd
              4 : JRE 1.7.0 Windows 7 amd64-64 build 20130711_156087 (pwa6470sr1-20130715_01 SR1))
```

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
```

Using jextract:

Use the **jextract** utility to process system dumps.

For an analysis of core 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 provided in the SDK to collect these files. You must run **jextract** using the same SDK level, on the same system that produced the system dump. The **jextract** utility compresses the dump, executable files, and libraries into a single `.zip` file for use in subsequent problem diagnosis.

For Java 7 SDKs on Windows and z/OS platforms, you do not need to run the **jextract** utility.

For Java 6 and Java 5.0 SDKs, containing versions of the IBM J9 virtual machine before V2.6, you must still run the **jextract** utility for all platforms.

When a core file is generated, run the **jextract** utility against the core file with the following syntax:

```
jextract <core file name> [<zip_file>]
```

to generate a compressed file in the current directory, containing the dump and the required executable file and libraries. The **jextract** utility is in the directory `sdk/jre/bin`. If you run **jextract** on a JVM level that is different from the one on which the dump was produced you see the following messages:

```
J9RAS.buildID is incorrect (found e8801ed67d21c6be, expecting eb4173107d21c673).
This version of jextract is incompatible with this dump.
Failure detected during jextract, see previous message(s).
```

Remember: If you are analyzing a dump from a JVM that used **-Xcompressedrefs**, use **-J-Xcompressedrefs** to run **jextract** using compressed references. See "Compressed references" on page 27 for more information about compressed references.

The contents of the `.zip` file produced and the contents of the XML are subject to change. You are advised not to design tools based on the contents of these files.

On z/OS, you can copy the dump to an HFS file and pass that as input to **jextract**. Alternatively you can pass a fully qualified MVS data set name as input to **jextract**. **jextract** is unable to read data sets larger than 2 GB directly using a 31 bit JVM and so you must use COPYDUMP first or move the dump to HFS. An example of the **jextract** command is:

```
> jextract USER1.JVM.TDUMP.SSHD6.D070430.T092211
Loading dump file...
Read memory image from USER1.JVM.TDUMP.SSHD6.D070430.T092211
VM set to 10BA5028
```

```
Dumping JExtract file to USER1.JVM.TDUMP.SSHD6.D070430.T092211.xml
<!-- extracting gpf state -->
...
Finished writing JExtract file in 5308ms
Creating archive file: USER1.JVM.TDUMP.SSHD6.D070430.T092211.zip
Adding "USER1.JVM.TDUMP.SSHD6.D070430.T092211" to archive
Adding "USER1.JVM.TDUMP.SSHD6.D070430.T092211.xml" to archive
Adding "/u/build/sdk/jre/lib/J9TraceFormat.dat" to archive
jextract complete.
```

This produces a compressed (`.zip`) file in the current HFS directory.

Problems to tackle with the dump viewer:

Dumps of JVM processes can arise either when you use the **-Xdump** option on the command line or when the JVM is not in control (such as user-initiated dumps).

jdmpview is most useful in diagnosing customer-type problems and problems with the class libraries. A typical scenario is OutOfMemoryError exceptions in customer applications.

For problems involving gpfs, ABENDS, SIGSEVs, and similar problems, you will obtain more information by using a system debugger (gdb) with the dump file. The syntax for the gdb command is

```
gdb <full_java_path> <system_dump_file>
```

For example:

```
gdb /sdk/jre/bin/java core.20060808.173312.9702.dmp
```

jdmpview can still provide useful information when used alone. Because **jdmpview** allows you to observe stacks and objects, the tool enables introspection into a Java program in the same way as a Java debugger. It allows you to examine objects, follow reference chains and observe Java stack contents. The main difference (other than the user interface) is that the program state is frozen; thus no stepping can occur. However, this allows you to take periodic program snapshots and perform analysis to see what is happening at different times.

Working with dumps containing multiple JVMs:

On z/OS, a system dump can contain multiple address spaces, and an address space can contain multiple JVMs. You can work with these dumps by using the **jdmpview** command, which separates the dump into contexts.

When you start the **jdmpview** tool, or run the **context** command, the tool shows 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 1.7.0 z/OS s390x-64 build 20111017_75924 (pmz6460_26-20111028_01)
6 : ASID: 0x73 EDB: 0x8004053a0 : JRE 1.7.0 z/OS s390x-64 build 20111017_75924 (pmz6460_26-20111028_01)
7 : ASID: 0x73 EDB: 0x4a7bd9e8 : No JRE
8 : ASID: 0xffff : No JRE : No JRE
```

Each address space (ASID) is listed as a separate context. Each JVM is also listed as a separate context. In the example, contexts `5` and `6`, in the address space `0x73`, are separate JVMs.

The current context is indicated by an asterisk (*), and by the prompt (`CTX:5>` in this example). If you enter a command at the prompt, the action is applied to the current context. You can switch between contexts by entering `context n`, where *n* is the context that you want to switch to. For example:

```
CTX:5> context 6
CTX: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 3.26.45, using DTFJ version 1.10.26062
Loading image from DTFJ...

For a list of commands, type "help"; for how to use "help", type "help help"
Available contexts (* = currently selected context) :
```

```
Source : file:/home/test/core.20120228.113859.14043.0001.dmp
        *0 : PID: 14073 : JRE 1.7.0 Linux ppc64-64 build 20120216_102976 (pxp6470sr1-20120221_01(SR1))
> info system

Machine OS:     Linux
Machine name:   madras
Machine IP address(es):
                9.20.88.155
System memory:  8269398016

Java version :

JRE 1.7.0 Linux ppc64-64 build 20120216_102976 (pxp6470sr1-20120221_01(SR1) )

> info proc

          Native thread IDs:
           14044 14073

          Command line arguments
           sdk/jre/bin/java -Xdump:system:events=vmstop -version

          JIT was enabled for this runtime
           AOT enabled, FSD enabled, HCR disabled, JIT enabled

          Environment variables:
           LESSKEY=/etc/lesskey.bin
           LESS_ADVANCED_PREPROCESSOR=no
           HOSTTYPE=ppc64
           CSHEDIT=emacs
           G_BROKEN_FILENAMES=1
           LESSOPEN=lessopen.sh %s
           MINICOM=-c on
           PATH=/home/test/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/X11R6/bin:/usr/lib/mit/bin:/usr/lib/mit/sbin
           INFODIR=/usr/local/info:/usr/share/info:/usr/info
```

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
- 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*>]
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.

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
Displays a list of all memory segments in the address space: Start address and size.

info heap [*|<*heap_name*>]
If no parameters are passed to this command, the heap names and heap sections are shown.

Using either "*" or a heap name shows the following information about all heaps or the specified heap:
- heap name
- (heap size and occupancy)
- heap sections
  - section name
  - section size
  - whether the section is shared
  - whether the section is executable
  - whether the section is read only

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 9. 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.

Example session:

This example session illustrates a selection of the commands available and their use.

In the example session, some lines have been removed for clarity (and terseness).

User input is prefaced by a greater than symbol (>).

```
TLBA82ME:/team/test/mz64$  sdk/bin/jdmpview -core J9BUILD.TEST.DMP.X001
DTFJView version 4.26.58, using DTFJ version 1.10.26068
Loading image from DTFJ...

For a list of commands, type "help"; for how to use "help", type "help help"
Available contexts (* = currently selected context) :

Source : file:///team/test/J9BUILD.TEST.DMP.X001
         0 : ASID: 0x1 : No JRE : No JRE
         1 : ASID: 0x4 : No JRE : No JRE
         2 : ASID: 0x7 : No JRE : No JRE
        *3 : ASID: 0x6b EDB: 0x4800105570 : JRE 1.7.0 z/OS s390x-64 build 20121116_128663 (pmz6470sr4-20121119_01(SR4))
         4 : ASID: 0x40404040 : No JRE : No JRE

> help
+                                       displays the next section of memory in hexdump-like format
-                                       displays the previous section of memory in hexdump-like format
cd                                      changes the current working directory, used for log files
close                   [context id]    closes the connection to a core file
context                 [ID|asid ID]    switch to the selected context
deadlock                                displays information about deadlocks if there are any
exit                                    Exit the application
find                                    searches memory for a given string
findnext                                finds the next instance of the last string passed to "find"
findptr                                 searches memory for the given pointer
heapdump                                generates a PHD or classic format heapdump
help                    [command name]  displays list of commands or help for a specific command
```

```
hexdump                                    outputs a section of memory in a hexdump-like format
info                       <component>     Information about the specified component
info class                 <Java class name>  Provides information about the specified Java class
info heap                  [*|heap name]   Displays information about Java heaps
info jitm                                  Displays JIT'ed methods and their addresses
info lock                                  outputs a list of system monitors and locked objects
info mmap                                  Outputs a list of all memory segments in the address space
info mod                                   outputs module information
info proc                                  shortened form of info process
info process                               displays threads, command line arguments, environment
info sym                                   an alias for 'mod'
info sys                                   shortened form of info system
info system                                displays information about the system the core dump is from
info thread                                displays information about Java and native threads
log                        [name level]    display and control instances of java.util.logging.Logger
open                       [path to core or zip] opens the specified core or zip file
plugins                                    Plugin management commands
                      list                 Show the list of loaded plugins for the current context
                    reload                 Reload plugins for the current context
                  showpath                 Show the DTFJ View plugin search path for the current context
                   setpath                 Set the DTFJ View plugin search path for the current context
pwd                                        displays the current working directory
quit                                       Exit the application
set                        [logging|heapdump]  Sets options for the specified command
set heapdump                               configures heapdump format, filename and multiple heap support
set logging                                configures several logging-related parameters, starts/stops logging
                        on                 turn on logging
                       off                 turn off logging
                      file                 turn on logging
                 overwrite                 controls the overwriting of log files
show                       [logging|heapdump]  Displays the current set options for a command
show heapdump                              displays heapdump settings
show logging                               shows the current logging options
whatis                     [hex address]   gives information about what is stored at the given memory address
x/d                        <hex address>   displays the integer at the specified address
x/j                        <object address> [class name] displays information about a particular object or all objects of a class
x/k                        <hex address>   displays the specified memory section as if it were a stack frame parameters
x/x                        <hex address>   displays the hex value of the bytes at the specified address

> set logging file log.txt
logging turned on; outputting to "/team/test/log.txt"

> info system

Machine OS:     z/OS
Machine name:   TLBA82ME
Machine IP address(es):
                9.26.177.7
System memory:  2147483648

Java version:
JRE 1.7.0 z/OS s390x-64 build 20121116_128663 (pmz6470sr4-20121119_01(SR4) )

> info thread *
native threads for address space
 process id: 16778998

  thread id: 0x25dc8700
   registers:
    PSW    = 0x078d1400a62831a2   R0     = 0x0000000000000000   R1     = 0x00000000263b1048   R2     = 0x00000000263b1860
    R3     = 0x00000000263b1048   R4     = 0x00000048109fbe80   R5     = 0x0000000000000000   R6     = 0x00000000262830e0
    R7     = 0x000000002629b25a   R8     = 0x00000000262830f2   R9     = 0x00000048109fe380   R10    = 0x00000048086240e0
    R11    = 0x000000002629bb10   R12    = 0x000000485312c790   R13    = 0x00000000263b1048   R14    = 0x0000000026283244
    R15    = 0x000000007f656f00
   native stack sections:
    0x48109f4000 to 0x4810a00000 (length 0xc000)
   native stack frames:
    bp: 0x00000048109fbe80 pc: 0x00000000262831a2 ExtraSymbolsModule::_TDUMP+0xc2
    bp: 0x00000048109fbf20 pc: 0x000000002629b258 /team/sdk/jre/lib/s390x/default/libj9prt26.so::tdump_wrapper+0x530
    bp: 0x00000048109fc960 pc: 0x000000002629a79a /team/sdk/jre/lib/s390x/default/libj9prt26.so::j9dump_create+0x4a
    bp: 0x00000048109fd060 pc: 0x0000000026321afa /team/sdk/jre/lib/s390x/default/libj9dmp26.so::doSystemDump+0x392
    bp: 0x00000048109fd160 pc: 0x0000000026322564 /team/sdk/jre/lib/s390x/default/libj9dmp26.so::protectedDumpFunction+0x2c
    bp: 0x00000048109fd260 pc: 0x00000000262b2fce /team/sdk/jre/lib/s390x/default/libj9prt26.so::j9sig_protect+0x76e
    bp: 0x00000048109fda60 pc: 0x0000000026325cea /team/sdk/jre/lib/s390x/default/libj9dmp26.so::runDumpAgent+0x442
    bp: 0x00000048109fdfa0 pc: 0x0000000026354e8e /team/sdk/jre/lib/s390x/default/libj9dmp26.so::triggerDumpAgents+0x616
    bp: 0x00000048109fe360 pc: 0x0000000026354086 /team/sdk/jre/lib/s390x/default/libj9dmp26.so::rasDumpHookVmShutdown+0xa6
    bp: 0x00000048109fe560 pc: 0x000000002617cdf0 /team/sdk/jre/lib/s390x/default/libj9vm26.so::protectedDestroyJavaVM+0x358
    bp: 0x00000048109fe6e0 pc: 0x00000000262b2fce /team/sdk/jre/lib/s390x/default/libj9prt26.so::j9sig_protect+0x76e
    bp: 0x00000048109feee0 pc: 0x000000002617e3e4 /team/sdk/jre/lib/s390x/default/libj9vm26.so::DestroyJavaVM+0x3b4
    bp: 0x00000048109ff020 pc: 0x00000000260b3ba4 /team/sdk/jre/lib/s390x/default/libjvm.so::DestroyJavaVM+0x3c
    bp: 0x00000048109ff120 pc: 0x000000002603751c libjli.so::JavaMain+0x754
   properties:
    PSW:31: 0x70c1000809afb42
    EDB: 0x4800105570
    Task Completion Code: 0x0
    CAA CEL level: 0x19
    CAA: 0x4810bfec60
    PTHREADID: 0x25dc870000000003
    TCB: 0x6bfcf0
    Stack direction: up
```

```
   associated Java thread:
    name:          DestroyJavaVM helper thread
    Thread object: java/lang/Thread @ 0x4839258a18
    Priority:      5
    Thread.State:  RUNNABLE
    JVMTI state:   ALIVE RUNNABLE
    Java stack frames: <no frames to print>
===Lines Removed===

    thread id: 0x25dd3100
   registers:
    PSW    = 0x078d0401a5ada6d4   R0     = 0x0000000000000001   R1     = 0x000000007f2e2f30   R2     = 0x0000004808710a50
    R3     = 0x0000000027f669a0   R4     = 0x000000487c3fdf40   R5     = 0x0000000025add868   R6     = 0x000000480870a770
    R7     = 0x0000004808710478   R8     = 0x000000007f2e2f30   R9     = 0x0000000000000000   R10    = 0x00000048531bc188
    R11    = 0x0000004800105570   R12    = 0x000000487c5fec60   R13    = 0x0000000000000000   R14    = 0x0000004808760 1f0
    R15    = 0x00000048806bde00
   native stack sections:
    0x487c3fc000 to 0x487c400000 (length 0x4000)
   native stack frames:
    bp: 0x000000487c3fdf40 pc: 0x0000000025ada6d4 ExtraSymbolsModule::CEEOPCW+0x2014
    bp: 0x000000487c3fe320 pc: 0x0000000025d7caf2 ExtraSymbolsModule::pthread_cond_wait+0x152
    bp: 0x000000487c3fe4a0 pc: 0x0000000026224082 libj9thr26.so::monitor_wait_original+0xb42
    bp: 0x000000487c3fe660 pc: 0x0000000026228430 libj9thr26.so::j9thread_monitor_wait+0x70
    bp: 0x000000487c3fe760 pc: 0x00000000278e7202 ExtraSymbolsModule::MM_ParallelDispatcher::slaveEntryPoint(MM_EnvironmentModron*)+0x82
    bp: 0x000000487c3fe860 pc: 0x00000000278e6f0a ExtraSymbolsModule::dispatcher_thread_proc2(J9PortLibrary*,void*)+0x13a
    bp: 0x000000487c3fe960 pc: 0x00000000262b2fce /team/sdk/jre/lib/s390x/default/libj9prt26.so::j9sig_protect+0x76e
    bp: 0x000000487c3ff160 pc: 0x00000000278e7158 ExtraSymbolsModule::dispatcher_thread_proc+0x58
    bp: 0x000000487c3ff260 pc: 0x0000000026226aa2 libj9thr26.so::thread_wrapper+0x562
   properties:
    EDB: 0x4800105570
    Task Completion Code: 0x0
    CAA CEL level: 0x19
    PSW:64: 0x78d0401a5ada6d4
    CAA: 0x487c5fec60
    PTHREADID: 0x25dd31000000000c
    TCB: 0x6bde88
    Stack direction: up
   associated Java thread:
    name:          GC Slave
    Thread object: java/lang/Thread @ 0x48392539f0
    Priority:      5
    Thread.State:  WAITING
    JVMTI state:   ALIVE WAITING WAITING_INDEFINITELY IN_OBJECT_WAIT
      waiting to be notified on: "MM_ParallelDispatcher::slaveThread" with ID 0x480862c120 owner name: <unowned>
    Java stack frames: <no frames to print>
===Lines Removed===


> info class java/lang/String
name = java/lang/String

        ID = 0x481908f300    superID = 0x483a1f6700
        classLoader = 0x48392807f0    modifiers: public final

        number of instances:     2668
        total size of instances: 85376 bytes

Inheritance chain....

        java/lang/Object
           java/lang/String

Fields......

          static fields for "java/lang/String"
            private static final long serialVersionUID = -6849794470754667710 (0xa0f0a4387a3bb342)
            public static final java.util.Comparator CASE_INSENSITIVE_ORDER = <object> @ 0x4839280318
            private static final char[] ascii = <object> @ 0x4839280328
            private static String[] stringArray = <object> @ 0x4839280438
            private static final int stringArraySize = 10 (0xa)
            static boolean enableCopy = false
            private static int seed = -1341042903 (0xffffffffb0114f29)
            private static char[] startCombiningAbove = <object> @ 0x4819311288
            private static char[] endCombiningAbove = <object> @ 0x4819311318
            private static final char[] upperValues = <object> @ 0x48193113a8
            private static final java.io.ObjectStreamField[] serialPersistentFields = <object> @ 0x4839280498

          non-static fields for "java/lang/String"
            private final char[] value
            private final int offset
            private final int count
            private int hashCode
            private int hashCode32


Methods......

Bytecode range(s): :  private static native int getSeed()
Bytecode range(s): 4839ce5e10 -- 4839ce5e26:  public void <init>()
Bytecode range(s): 4839ce5e50 -- 4839ce5e95:  private void <init>(String, char)
```

```
Bytecode range(s): 4839ce5ed8 -- 4839ce5ee1:  public void <init>(byte[])
Bytecode range(s): 4839ce5f08 -- 4839ce5f12:  public void <init>(byte[], int)
Bytecode range(s): 4839ce5f44 -- 4839ce5f7d:  public void <init>(byte[], int, int)
===Lines Removed===

> whatis 0x4839280438
        heap #1 - name: Generational@4839e11ef0
                0x4839280438 is within heap segment: 4839240000 -- 4839300000
                0x4839280438 is the start of an object of type [Ljava/lang/String;
```

jdmpview commands quick reference:

A short list of the commands you use with **jdmpview**.

The following table shows the jdmpview - quick reference:

Command	Sub-command	Description
help		Displays a list of commands or help for a specific command.
info		
	thread	Displays information about Java and native threads.
	system	Displays information about the system the core dump is from.
	class	Displays the inheritance chain and other data for a given class.
	proc	Displays threads, command line arguments, environment variables, and shared modules of current process.
	jitm	Displays JIT compiled methods and their addresses.
	lock	Displays a list of available monitors and locked objects.
	sym	Displays a list of available modules.
	mmap	Displays a list of all memory segments in the address space.
	heap	Displays information about all heaps or the specified heap.
heapdump		Generates a Heapdump to a file. You can select which Java heaps should be dumped by listing the heap names, separated by spaces.
hexdump		Displays a section of memory in a hexdump-like format.
+		Displays the next section of memory in hexdump-like format.
-		Displays the previous section of memory in hexdump-like format.
whatis		Displays information about what is stored at the given memory address.
find		Searches memory for a given string.
findnext		Finds the next instance of the last string passed to “find”.
findptr		Searches memory for the given pointer.
x/ (examine)		**Examine** works like **x/** in gdb, including the use of defaults: passes the number of items to display and the unit size (`b` for byte (8-bit), `h` for half word (16-bit), `w` for word (32-bit), `g` for giant word (64-bit)) to the sub-command. For example `x/12bd`.
	J	Displays information about a particular object or all objects of a class.

Command	Sub-command	Description
	D	Displays the integer at the specified address.
	X	Displays the hex value of the bytes at the specified address.
	K	Displays the specified memory section as if it were a stack frame.
deadlock		Displays information about deadlocks if there are any set.
set heapdump		Configures Heapdump generation settings.
set logging		Configures logging settings, starts logging, or stops logging. This allows the results of commands to be logged to a file.
show heapdump		Displays the current values of heapdump settings.
show logging		Displays the current values of logging settings.
cd		Changes the current working directory, used for log files.
pwd		Displays the current working directory.
quit		Exits the core file viewing tool; any log files that are currently open are closed before the tool exits.

Tracing Java applications and the JVM

JVM trace is a trace facility that is provided in all IBM-supplied JVMs with minimal affect on performance. Trace data can be output in human-readable or in compressed binary formats. The JVM provides a tool to process and convert the compressed binary data and into a readable format.

JVM trace data might contain application data, including the contents of Java objects. If you require a dump that does not contain this application data, see “Using Javadump” on page 240 or “Using Heapdump” on page 262.

Tracing is enabled by default, together with a small set of trace points going to memory buffers. You can enable tracepoints at run time by using levels, components, group names, or individual tracepoint identifiers.

This chapter describes JVM trace in:

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

What can be traced?

You can trace JVM internals, applications, and Java method or any combination of those.

JVM internals
The IBM Virtual Machine for Java is extensively instrumented with tracepoints for trace. Interpretation of this trace data requires knowledge of the internal operation of the JVM, and is provided to diagnose JVM problems.

No guarantee is given that tracepoints will not vary from release to release and from platform to platform.

Applications
JVM trace contains an application trace facility that allows tracepoints to be placed in Java code to provide trace data that will be combined with the other forms of trace. There is an API in the com.ibm.jvm.Trace class to support this. Note that an instrumented Java application runs only on an IBM-supplied JVM.

Java methods
You can trace entry to and exit from Java methods run by the JVM. You can select method trace by classname, method name, or both. You can use wildcards to create complex method selections.

JVM trace can produce large amounts of data in a very short time. Before running trace, think carefully about what information you need to solve the problem. In many cases, where you need only the trace information that is produced shortly before the problem occurs, consider using the **wrap** option. In many cases, just use internal trace with an increased buffer size and snap the trace when the problem occurs. If the problem results in a thread stack dump or operating system signal or exception, trace buffers are snapped automatically to a file that is in the current directory. The file is called: Snap*nnnn.yyyymmdd.hhmmssth.process*.trc.

You must also think carefully about which components need to be traced and what level of tracing is required. For example, if you are tracing a suspected shared classes problem, it might be enough to trace all components at level 1, and j9shr at level 9, while maximal can be used to show parameters and other information for the failing component.

Types of tracepoint

There are two types of tracepoints inside the JVM: regular and auxiliary.

Regular tracepoints

Regular tracepoints include:

- method tracepoints
- application tracepoints
- data tracepoints inside the JVM
- data tracepoints inside class libraries

You can display regular tracepoint data on the screen or save the data to a file. You can also use command line options to trigger specific actions when regular tracepoints fire. See the section “Detailed descriptions of trace options” on page 294 for more information about command line options.

Auxiliary tracepoints

Auxiliary tracepoints are a special type of tracepoint that can be fired only when another tracepoint is being processed. An example of auxiliary tracepoints are the tracepoints containing the stack frame information produced by the **jstacktrace -Xtrace:trigger** command. You cannot control where auxiliary tracepoint data is sent and you cannot set triggers on auxiliary tracepoints. Auxiliary tracepoint data is sent to the same destination as the tracepoint that caused them to be generated.

Default tracing

By default, the equivalent of the following trace command line is always available in the JVM:

```
-Xtrace:maximal=all{level1},exception=j9mm{gclogger}
```

The data generated by the tracepoints is continuously captured in wrapping memory buffers for each thread. (For information about specific options, see "Detailed descriptions of trace options" on page 294.)

You can find tracepoint information in the following diagnostics data:

- System memory dumps, extracted by using **jdmpview**.
- Snap traces, generated when the JVM encounters a problem or an output file is specified. "Using dump agents" on page 221 describes more ways to create a snap trace.
- For exception trace only, in Javadumps.

Default memory management tracing

The default trace options are designed to ensure that Javadumps always contain a record of the most recent memory management history, regardless of how much work the JVM has performed since the garbage collection cycle was last called.

The **exception=j9mm{gclogger}** clause of the default trace set specifies that a history of garbage collection cycles that have occurred in the JVM is continuously recorded. The **gclogger** group of tracepoints in the j9mm component constitutes a set of tracepoints that record a snapshot of each garbage collection cycle. These tracepoints are recorded in their own separate buffer, called the exception buffer. The effect is that the tracepoints are not overwritten by the higher frequency tracepoints of the JVM.

The **GC History** section of the Javadump is based on the information in the exception buffer. If a garbage collection cycle has occurred in a traced JVM, the Javadump probably contains a **GC History** section.

Default assertion tracing

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

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
```

Assertions are special trace points. They can be enabled or disabled by using the standard trace command-line options. See "Controlling the trace" on page 293 for more details.

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.

Where does the data go?

Trace data can be written to a number of locations.

Trace data can go into:

- Memory buffers that can be dumped or snapped when a problem occurs
- One or more files that are using buffered I/O
- An external agent in real time
- stderr in real time
- Any combination of the other items in this list

Writing trace data to memory buffers:

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

Buffers are allocated on a per-thread principle. This principle removes contention between threads, and prevents trace data for an individual thread from being mixed in with trace data from other threads. For example, if one particular thread is not being dispatched, its trace information is still available when the buffers are dumped or snapped. Use the **-Xtrace:buffers=**<*size*> option to control the size of the buffer allocated to each thread. Buffers allocated to a thread are discarded when that thread terminates.

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

To examine the trace data captured in these memory buffers, you must snap or dump the data, then format the buffers.

Snapping buffers

Under default conditions, a running JVM collects a small amount of trace data in special wraparound buffers. This data is sent to a snap trace file under certain conditions:

- An uncaught OutOfMemoryError occurs.
- An operating system signal or exception occurs.
- The com.ibm.jvm.Trace.snap() Java API is called.
- The JVMRI TraceSnap function is called.

The resulting snap trace file is placed into the current working directory, with a name in the format `Snap`*`nnnn.yyyymmdd.hhmmssth.process.`*`trc`, where *nnnn* is a sequence number reset to 0001 at JVM startup, *yyyymmdd* is the current date, *hhmmssth* is the current time, and *process* is the process identifier. This file is in a binary format, and requires the use of the supplied trace formatter so that you can read it.

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

Extracting buffers from system dump

You can extract the buffers from a system dump core file by using the Dump Viewer.

Writing trace data to a file:

You can write trace data to a file continuously as an extension to the in-storage trace, but, instead of one buffer per thread, at least two buffers per thread are allocated, and the data is written to the file before wrapping can occur.

This allocation allows the thread to continue to run while a full trace buffer is written to disk. Depending on trace volume, buffer size, and the bandwidth of the output device, multiple buffers might be allocated to a given thread to keep pace with trace data that is being generated.

A thread is never stopped to allow trace buffers to be written. If the rate of trace data generation greatly exceeds the speed of the output device, excessive memory usage might occur and cause out-of-memory conditions. To prevent this, use the **nodynamic** option of the **buffers** trace option. For long-running trace runs, a **wrap** option is available to limit the file to a given size. It is also possible to create a sequence of files when the trace output will move back to the first file once the sequence of files are full. See the **output** option for details. You must use the trace formatter to format trace data from the file.

Because trace data is buffered, if the JVM does not exit normally, residual trace buffers might not be flushed to the file. If the JVM encounters an unrecoverable error, the buffers can be extracted from a system dump if that is available. When a snap file is created, all available buffers are always written to it.

External tracing:

You can route trace to an agent by using JVMRI TraceRegister.

This mechanism allows a callback routine to be called immediately when any of the selected tracepoints is found without buffering the trace results. The trace data is in raw binary form. Further details can be found in the JVMRI section.

Tracing to stderr:

For reduced volume or non-performance-critical tracing, the trace data can be formatted and routed to stderr immediately without buffering.

For more information, see "Using method trace" on page 316.

Trace combinations:

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

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

Controlling the trace

You have several ways by which you can control the trace.

You can control the trace in several ways by using:

- The **-Xtrace** options when launching the JVM, including trace trigger events
- A trace properties file
- com.ibm.jvm.Trace API
- JVMTI and JVMRI from an external agent

Note:

1. The specification of trace options is cumulative. Multiple **-Xtrace** options are accepted on the command line and they are processed in order starting with the option that is closest to the **-Xtrace** string. Each option adds to the previous options (and to the default options), as if they had all been specified in one long comma-separated list in a single option. This cumulative specification is consistent with the related **-Xdump** option processing.
2. Some trace options are enabled by default. For more information, see "Default tracing" on page 290. To disable the defaults, use the **-Xtrace:none** option.
3. Many diagnostic tools start a JVM. When using the **IBM_JAVA_OPTIONS** environment variable trace to a file, starting a diagnostic tool might overwrite the trace data generated from your application. Use the command-line tracing options or add %d, %p or %t to the trace file name to prevent this from happening. See "Detailed descriptions of trace options" on page 294 for the appropriate trace option description.

Specifying trace options:

The preferred way to control trace is through trace options that you specify by using the **-Xtrace** option on the launcher command line, or by using the **IBM_JAVA_OPTIONS** environment variable.

Some trace options have the form `<name>` and others are of the form `<name>=<value>`, where `<name>` is case-sensitive. Except where stated, `<value>` is not case-sensitive; the exceptions to this rule are file names on some platforms, class names, and method names.

If an option value contains commas, it must be enclosed in braces. For example:

```
methods={java/lang/*,com/ibm/*}
```

Note: The requirement to use braces applies only to options specified on the command line. You do not need to use braces for options specified in a properties file.

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

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

To switch off all tracepoints, use this option:

```
java -Xtrace:none=all
```

If you specify other tracepoints without specifying **-Xtrace:none**, the tracepoints are added to the default set.

When you use the **IBM_JAVA_OPTIONS** environment variable, use this syntax:

```
set IBM_JAVA_OPTIONS=-Xtrace:<name>,<another_name>=<value>
```

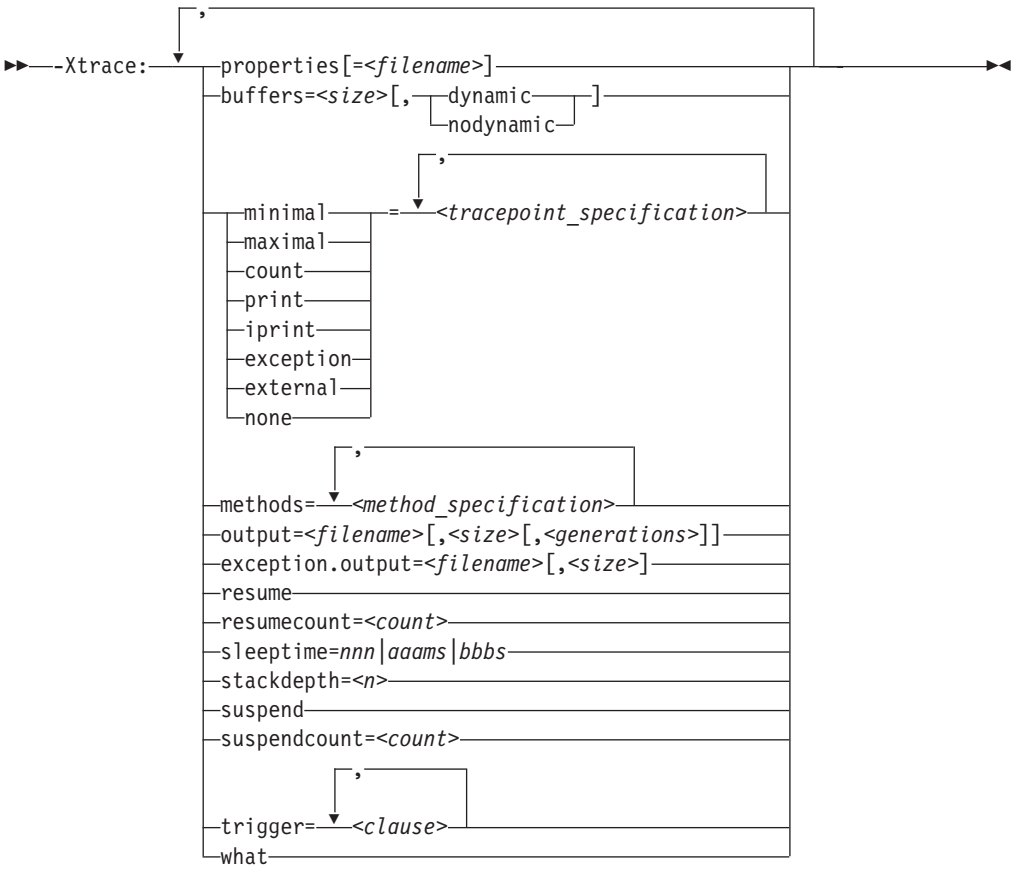
or

```
export IBM_JAVA_OPTIONS=-Xtrace:<name>,<another_name>=<value>
```

Detailed descriptions of trace options:

The options are processed in the sequence in which they are described here.

-Xtrace command-line option syntax



properties[=<filename>]:

You can use properties files to control trace. A properties file saves typing and, over time, causes a library of these files to be created. Each file is tailored to solving problems in a particular area.

This trace option allows you to specify in a file any of the other trace options, thereby reducing the length of the invocation command-line. The format of the file is a flat ASCII file that contains trace options. If *<filename>* is not specified, a default name of `IBMTRACE.properties` is searched for in the current directory. Nesting is not supported; that is, the file cannot contain a `properties` option. If any error is found when the file is accessed, JVM initialization fails with an explanatory error message and return code. All the options that are in the file are processed in the sequence in which they are stored in the file, before the next option that is obtained through the normal mechanism is processed. Therefore, a command-line property always overrides a property that is in the file.

An existing restriction means that you cannot leave properties that have the form **<name>=<value>** to default if they are specified in the property file; that is, you must specify a value, for example `maximal=all`.

Another restriction means that properties files are sensitive to white space. Do not add white space before, after, or within the trace options.

You can make comments as follows:

```
// This is a comment. Note that it starts in column 1
```

Examples

- Use `IBMTRACE.properties` in the current directory:

  ```
  -Xtrace:properties
  ```

  – Use `trace.prop` in the current directory:

    ```
    -Xtrace:properties=trace.prop
    ```

  – Use `c:\trc\gc\trace.props`:

    ```
    -Xtrace:properties=c:\trc\gc\trace.props
    ```

Here is an example property file:

```
minimal=all
// maximal=j9mm
maximal=j9shr
buffers=20k
output=c:\traces\classloader.trc
print=tpnid(j9vm.23-25)
```

buffers=nnnk|nnnm[,dynamic|nodynamic]:

You can modify the size of the buffers to change how much diagnostic output is provided in a snap dump. This buffer is allocated for each thread that makes trace entries.

The trace option can be specified in two ways:

- **buffers=dynamic|nodynamic**
- **buffers=nnnk|nnnm[,dynamic|nodynamic]**

If external trace is enabled, the number of buffers is doubled; that is, each thread allocates two or more buffers. The same buffer size is used for state and exception tracing, but, in this case, buffers are allocated globally. The default is 8 KB per thread.

The **dynamic** and **nodynamic** options have meaning only when tracing to an output file. If **dynamic** is specified, buffers are allocated as needed to match the rate of trace data generation to the output media. Conversely, if **nodynamic** is specified, a maximum of two buffers per thread is allocated. The default is **dynamic**. The dynamic option is effective only when you are tracing to an output file.

Note: If **nodynamic** is specified, you might lose trace data if the volume of trace data that is produced exceeds the bandwidth of the trace output file. Message `UTE115` is issued when the first trace entry is lost, and message `UTE018` is issued at JVM termination.

Examples

- Dynamic buffering with increased buffer size of 2 MB per thread:

  ```
  -Xtrace:buffers=2m
  ```

  or in a properties file:

  ```
  buffers=2m
  ```

- Trace buffers limited to two buffers per thread, each of 128 KB:

  ```
  -Xtrace:buffers={128k,nodynamic}
  ```

or in a properties file:

```
buffers=128k,nodynamic
```

- Trace using default buffer size of 8 KB, limited to two buffers per thread:

```
-Xtrace:buffers=nodynamic
```

or in a properties file:

```
buffers=nodynamic
```

Options that control tracepoint activation:

These options control which individual tracepoints are activated at run time and the implicit destination of the trace data.

In some cases, you must use them with other options. For example, if you specify maximal or minimal tracepoints, the trace data is put into memory buffers. If you are going to send the data to a file, you must use an output option to specify the destination filename.

minimal=[!]<*tracepoint_specification*>[,...]
maximal=[!]<*tracepoint_specification*>[,...]
count=[!]<*tracepoint_specification*>[,...]
print=[!]<*tracepoint_specification*>[,...]
iprint=[!]<*tracepoint_specification*>[,...]
exception=[!]<*tracepoint_specification*>[,...]
external=[!]<*tracepoint_specification*>[,...]
none[=<*tracepoint_specification*>[,...]]

Note that all these properties are independent of each other and can be mixed and matched in any way that you choose.

You must provide at least one tracepoint specification when using the **minimal**, **maximal**, **count**, **print**, **iprint**, **exception** and **external** options. In some older versions of the SDK the tracepoint specification defaults to 'all'.

Multiple statements of each type of trace are allowed and their effect is cumulative. To do this, you must use a trace properties file for multiple trace options of the same name.

minimal and maximal
: **minimal** and **maximal** trace data is placed into internal trace buffers that can then be written to a snap file or written to the files that are specified in an output trace option. The **minimal** option records only the timestamp and tracepoint identifier. When the trace is formatted, missing trace data is replaced with the characters "???" in the output file. The **maximal** option specifies that all associated data is traced. If a tracepoint is activated by both trace options, **maximal** trace data is produced. Note that these types of trace are completely independent from any types that follow them. For example, if the **minimal** option is specified, it does not affect a later option such as **print**.

count
: The **count** option requests that only a count of the selected tracepoints is kept. At JVM termination, all non-zero totals of tracepoints (sorted by tracepoint id) are written to a file, called utTrcCounters, in the current directory. This information is useful if you want to determine the overhead of particular tracepoints, but do not want to produce a large amount (GB) of trace data.

For example, to count the tracepoints used in the default trace configuration, use the following command:

```
-Xtrace:count=all{level1},count=j9mm{gclogger}
```

print
The **print** option causes the specified tracepoints to be routed to stderr in real time. The JVM tracepoints are formatted using `J9TraceFormat.dat`. The class library tracepoints are formatted by `TraceFormat.dat`. `J9TraceFormat.dat` and `TraceFormat.dat` are shipped in sdk/jre/lib and are automatically found by the run time environment.

iprint
The **iprint** option is the same as the **print** option, but uses indenting to format the trace.

exception
When **exception** trace is enabled, the trace data is collected in internal buffers that are separate from the normal buffers. These internal buffers can then be written to a snap file or written to the file that is specified in an **exception.output** option.

The **exception** option allows low-volume tracing in buffers and files that are distinct from the higher-volume information that **minimal** and **maximal** tracing have provided. In most cases, this information is exception-type data, but you can use this option to capture any trace data that you want.

This form of tracing is channeled through a single set of buffers, as opposed to the buffer-per-thread approach for normal trace, and buffer contention might occur if high volumes of trace data are collected. A difference exists in the <*tracepoint_specification*> defaults for exception tracing; see "Tracepoint specification" on page 299.

Note: The exception trace buffers are intended for low-volume tracing. By default, the exception trace buffers log garbage collection event tracepoints, see "Default tracing" on page 290. You can send additional tracepoints to the exception buffers or switch off the garbage collection tracepoints. Changing the exception trace buffers will alter the contents of the **GC History** section in any Javadumps.

Note: When **exception** trace is entered for an active tracepoint, the current thread id is checked against the previous caller's thread ID. If it is a different thread, or this is the first call to **exception** trace, a context tracepoint is put into the trace buffer first. This context tracepoint consists only of the current thread ID. This is necessary because of the single set of buffers for exception trace. (The formatter identifies all trace entries as coming from the `Exception trace pseudo thread` when it formats **exception** trace files.)

external
The **external** option channels trace data to registered trace listeners in real time. JVMRI is used to register or deregister as a trace listener. If no listeners are registered, this form of trace does nothing except waste machine cycles on each activated tracepoint.

none

-Xtrace:none prevents the trace engine from loading if it is the only trace option specified. However, if other **-Xtrace** options are on the command line, it is treated as the equivalent of **-Xtrace:none=all** and the trace engine will still be loaded.

If you specify other tracepoints without specifying **-Xtrace:none**, the tracepoints are added to the default set.

Examples

- Default options applied:

  ```
  java
  ```

- No effect apart from ensuring that the trace engine is loaded (which is the default behavior):

  ```
  java -Xtrace
  ```

- Trace engine is not loaded:

  ```
  java -Xtrace:none
  ```

- Trace engine is loaded, but no tracepoints are captured:

  ```
  java -Xtrace:none=all
  ```

- Default options applied, with the addition of printing for j9vm.209

  ```
  java -Xtrace:iprint=j9vm.209
  ```

- Default options applied, with the addition of printing for j9vm.209 and j9vm.210. Note the use of brackets when specifying multiple tracepoints.

  ```
  java -Xtrace:iprint={j9vm.209,j9vm.210}
  ```

- Printing for j9vm.209 only:

  ```
  java -Xtrace:none -Xtrace:iprint=j9vm.209
  ```

- Printing for j9vm.209 only:

  ```
  java -Xtrace:none,iprint=j9vm.209
  ```

- Default tracing for all components except j9vm, with printing for j9vm.209:

  ```
  java -Xtrace:none=j9vm,iprint=j9vm.209
  ```

- Default tracing for all components except j9vm, with printing for j9vm.209

  ```
  java -Xtrace:none=j9vm -Xtrace:iprint=j9vm.209
  ```

- No tracing for j9vm (none overrides iprint):

  ```
  java -Xtrace:iprint=j9vm.209,none=j9vm
  ```

Tracepoint specification:

You enable tracepoints by specifying *component* and *tracepoint*.

If no qualifier parameters are entered, all tracepoints are enabled, except for **exception.output** trace, where the default is **all {exception}**.

The <*tracepoint_specification*> is as follows:

```
[!]<component>[{<group>}] or [!]<component>[{<type>}] or [!]<tracepoint_id>[,<tracepoint_id>]
```

where:

! is a logical not. That is, the tracepoints that are in a specification starting with **!** are turned off.

<*component*>
is a Java component, as detailed in Table 10. To include all Java components, specify **all**.

Table 10. Java components

Component name	Description
avl	VM AVL tree support
io	Class library java.io native code

Table 10. Java components (continued)

Component name	Description
j9bcu	VM byte code utilities
j9bcverify	VM byte code verification
j9codertvm	VM byte code run time
j9dmp	VM dump
j9jcl	VM class libraries
j9jit	VM JIT interface
j9jvmti	VM JVMTI support
j9mm	VM memory management
j9prt	VM port library
j9scar	VM class library interface
j9shr	VM shared classes
j9trc	VM trace
j9util	VM utilities
j9vm	VM general
j9vmutil	VM utilities
j9vrb	VM verbose stack walker
map	VM mapped memory support
mt	Java methods (see note)
pool	VM storage pool support
rpc	VM RPC support
simplepool	VM storage pool support
sunvmi	VM class library interface

Note: When specifying the mt component you must also specify the `methods` option.

<group>
is a tracepoint group. A group is a set of tracepoints that are defined within a component, therefore each group is associated with one or more components as follows:

Table 11. Tracepoint groups and associated components

Component name or names	Group name	Description
mt	compiledMethods	A set of tracepoints that record compiled Java methods
mt	nativeMethods	A set of tracepoints that record Java native methods
mt	staticMethods	A set of tracepoints that record Java static methods

<type> is the tracepoint type. The following types are supported:
- Entry
- Exit
- Event
- Exception

- Mem

<tracepoint_id>
is the tracepoint identifier. The tracepoint identifier constitutes the component name of the tracepoint, followed by its integer number inside that component. For example, `j9mm.49`, `j9shr.20-29`, `j9vm.15`, To understand these numbers, see "Determining the tracepoint ID of a tracepoint" on page 312.

Some tracepoints can be both an exit and an exception; that is, the function ended with an error. If you specify either exit or exception, these tracepoints are included.

The following tracepoint specification used in Java 5.0 and earlier IBM SDKs is still supported:

```
[!]tpnid{<tracepoint_id>[,...]}
```

Examples

- All tracepoints:
  ```
  -Xtrace:maximal=all
  ```
- All tracepoints except `j9vrb` and `j9trc`:
  ```
  -Xtrace:minimal={all},minimal={!j9vrb,j9trc}
  ```
- All entry and exit tracepoints in `j9bcu`:
  ```
  -Xtrace:maximal={j9bcu{entry},j9bcu{exit}}
  ```
- All tracepoints in `j9mm` except tracepoints 20-30:
  ```
  -Xtrace:maximal=j9mm,maximal=!j9mm.20-30
  ```
- Tracepoints `j9prt.5` through `j9prt.15`:
  ```
  -Xtrace:print=j9prt.5-15
  ```
- All `j9trc` tracepoints:
  ```
  -Xtrace:count=j9trc
  ```
- All entry and exit tracepoints:
  ```
  -Xtrace:external={all{entry},all{exit}}
  ```

Trace levels:

Tracepoints have been assigned levels 0 through 9 that are based on the importance of the tracepoint.

A level 0 tracepoint is the most important. It is reserved for extraordinary events and errors. A level 9 tracepoint is in-depth component detail. To specify a given level of tracing, the level0 through level9 keywords are used. You can abbreviate these keywords to l0 through l9. For example, if level5 is selected, all tracepoints that have levels 0 through 5 are included. Level specifications do not apply to explicit tracepoint specifications that use the TPNID keyword.

The level is provided as a modifier to a component specification, for example:

```
-Xtrace:maximal={all{level5}}
```

or

```
-Xtrace:maximal={j9mm{L2},j9trc,j9bcu{level9},all{level1}}
```

In the first example, tracepoints that have a level of 5 or less are enabled for all components. In the second example, all level 1 tracepoints are enabled. All level2 tracepoints in j9mm are enabled. All tracepoints up to level 9 are enabled in j9bcu.

Note: The level applies only to the current component. If multiple trace selection components are found in a trace properties file, the level is reset to the default for each new component.

Level specifications do not apply to explicit tracepoint specifications that use the TPNID keyword.

When the **not** operator is specified, the level is inverted; that is, `!j9mm{level5}` disables all tracepoints of level 6 or greater for the j9mm component. For example:

```
-Xtrace:print={all},print={!j9trc{l5},j9mm{l6}}
```

enables trace for all components at level 9 (the default), but disables level 6 and higher for the locking component, and level 7 and higher for the storage component.

Examples

- Count the level zero and level one tracepoints matched:

  ```
  -Xtrace:count=all{L1}
  ```

- Produce maximal trace of all components at level 5 and j9mm at level 9:

  ```
  -Xtrace:maximal={all{level5},j9mm{L9}}
  ```

- Trace all components at level 6, but do not trace j9vrb at all, and do not trace the entry and exit tracepoints in the j9trc component:

  ```
  -Xtrace:minimal={all{l6}},minimal={!j9vrb,j9trc{entry},j9trc{exit}}
  ```

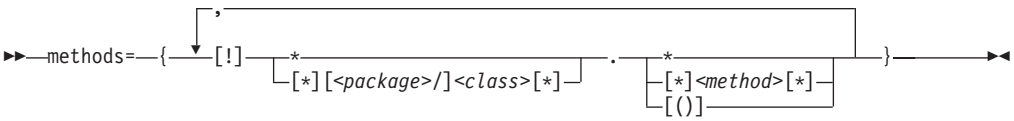
methods=<method_specification>[,<method_specification>]:

Using method trace provides a complete (and potentially large) diagnosis of code paths inside your application and the system classes. Use wild cards and filtering to control method trace so that you can focus on the sections of code that interest you.

Method trace can trace:

- Method entry
- Method exit

The `methods` parameter is defined as:



Where:

- The delimiter between parts of the package name is a forward slash, "/".
- The **!** in the methods parameter is a NOT operator that allows you to tell the JVM not to trace the specified method or methods.
- The parentheses, **()**, define whether or not to include method parameters in the trace.
- If a method specification includes any commas, the whole specification must be enclosed in braces, for example:

  ```
  -Xtrace:methods={java/lang/*,java/util/*},print=mt
  ```

- It might be necessary to enclose your command line in quotation marks to prevent the shell intercepting and fragmenting comma-separated command lines, for example:

  ```
  "-Xtrace:methods={java/lang/*,java/util/*},print=mt"
  ```

To output all method trace information to stderr, use:

-Xtrace:print=mt,methods=*.*
  Print method trace information for all methods to stderr.

-Xtrace:iprint=mt,methods=*.*
  Print method trace information for all methods to stderr using indentation.

To output method trace information in binary format, see "output=<*filename*>[,*size*m[,<*generations*>]]" on page 305.

Examples

- **Tracing entry and exit of all methods in a given class:**

  ```
  -Xtrace:methods={ReaderMain.*,java/lang/String.*},print=mt
  ```

  This traces all method entry and exit of the ReaderMain class in the default package and the java.lang.String class.
- **Tracing entry, exit and input parameters of all methods in a class:**

  ```
  -Xtrace:methods=ReaderMain.*(),print=mt
  ```

  This traces all method entry, exit, and input of the ReaderMain class in the default package.
- **Tracing all methods in a given package:**

  ```
  -Xtrace:methods=com/ibm/socket/*.*(),print=mt
  ```

  This traces all method entry, exit, and input of all classes in the package com.ibm.socket.
- **Multiple method trace:**

  ```
  -Xtrace:methods={Widget.*(),common/*},print=mt
  ```

  This traces all method entry, exit, and input in the Widget class in the default package and all method entry and exit in the common package.
- **Using the ! operator**

  ```
  -Xtrace:methods={ArticleUI.*,!ArticleUI.get*},print=mt
  ```

  This traces all methods in the ArticleUI class in the default package except those beginning with "get".
- **Tracing a specific method in a class**

  ```
  -Xtrace:print=mt,methods={java/lang/String.substring}
  ```

  This example traces entry and exit of the substring method of the java.lang.String class. If there is more than one method with the same name, they are all traced. You cannot filter method trace by the signature of the method.
- **Tracing the constructor of a class**

  ```
  -Xtrace:print=mt,methods={java/lang/String.<init>}
  ```

  This example traces entry and exit of the constructors of the java.lang.String class.

Example output

```
java "-Xtrace:methods={java/lang*.*},iprint=mt" HW
10:02:42.281*0x9e900      mt.4         > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.281 0x9e900      mt.4          > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.281 0x9e900      mt.4           > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.281 0x9e900      mt.4            > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.281 0x9e900      mt.10           < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.281 0x9e900      mt.10          < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.281 0x9e900      mt.4           > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.281 0x9e900      mt.10          < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.281 0x9e900      mt.10         < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.281 0x9e900      mt.4          > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.281 0x9e900      mt.4           > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.296 0x9e900      mt.10          < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.296 0x9e900      mt.10         < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.296 0x9e900      mt.4          > java/lang/String.<clinit>()V Compiled static method
10:02:42.296 0x9e900      mt.4           > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.296 0x9e900      mt.4            > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.296 0x9e900      mt.4             > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.296 0x9e900      mt.10            < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.296 0x9e900      mt.4             > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.328 0x9e900      mt.10            < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.328 0x9e900      mt.10           < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.328 0x9e900      mt.4            > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.328 0x9e900      mt.10           < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                              V Compiled static method
10:02:42.328 0x9e900      mt.4            > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.328 0x9e900      mt.10           < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                              V Compiled static method
10:02:42.328 0x9e900      mt.10          < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                              V Compiled static method
```

The output lines comprise of:

- `0x9e900`, the current **execenv** (execution environment). Because every JVM thread has its own **execenv**, you can regard **execenv** as a thread-id. All trace with the same **execenv** relates to a single thread.
- The individual tracepoint id in the mt component that collects and emits the data.
- The remaining fields show whether a method is being entered (>) or exited (<), followed by details of the method.

output=<filename>[,sizem[,<generations>]]:

Use the output option to send trace data to *<filename>*. If the file does not already exist, it is created automatically. If it does already exist, it is overwritten.

Optionally:

- You can limit the file to *size* MB, at which point it wraps to the beginning. If you do not limit the file, it grows indefinitely, until limited by disk space.
- If you want the final trace filename to contain today's date, the PID number that produced the trace, or the time, do one of the following steps as appropriate (see also the examples at the end of this section).
  - To include today's date (in "yyyymmdd" format) in the trace filename, specify "%d" as part of the *<filename>*.
  - To include the pidnumber of the process that is generating the tracefile, specify "%p" as part of the *<filename>*.
  - To include the time (in 24-hour hhmmss format) in the trace filename, specify "%t" as part of the *<filename>*.
- You can specify generations as a value 2 through 36. These values cause up to 36 files to be used in a round-robin way when each file reaches its size threshold. When a file needs to be reused, it is overwritten. If generations is specified, the filename must contain a "#" (hash, pound symbol), which will be substituted with its generation identifier, the sequence of which is 0 through 9 followed by A through Z.

Note: When tracing to a file, buffers for each thread are written when the buffer is full or when the JVM terminates. If a thread has been inactive for a period of time before JVM termination, what seems to be 'old' trace data is written to the file. When formatted, it then seems that trace data is missing from the other threads, but this is an unavoidable side-effect of the buffer-per-thread design. This effect becomes especially noticeable when you use the generation facility, and format individual earlier generations.

Examples

- Trace output goes to /u/traces/gc.problem; no size limit:
  ```
  -Xtrace:output=/u/traces/gc.problem,maximal=j9gc
  ```
- Output goes to trace and will wrap at 2 MB:
  ```
  -Xtrace:output={trace,2m},maximal=j9gc
  ```
- Output goes to gc0.trc, gc1.trc, gc2.trc, each 10 MB in size:
  ```
  -Xtrace:output={gc#.trc,10m,3},maximal=j9gc
  ```
- Output filename contains today's date in yyyymmdd format (for example, traceout.20041025.trc):
  ```
  -Xtrace:output=traceout.%d.trc,maximal=j9gc
  ```
- Output file contains the number of the process (the PID number) that generated it (for example, tracefrompid2112.trc):
  ```
  -Xtrace:output=tracefrompid%p.trc,maximal=j9gc
  ```
- Output filename contains the time in hhmmss format (for example, traceout.080312.trc):
  ```
  -Xtrace:output=traceout.%t.trc,maximal=j9gc
  ```

exception.output=<filename>[,nnnm]:

Use the exception option to redirect exception trace data to *<filename>*.

If the file does not already exist, it is created automatically. If it does already exist, it is overwritten. Optionally, you can limit the file to nnn MB, at which point it wraps nondestructively to the beginning. If you do not limit the file, it grows indefinitely, until limited by disk space.

Optionally, if you want the final trace filename to contain today's date, the PID number that produced the trace, or the time, do one of the following steps as appropriate (see also the examples at the end of this section).

- To include today's date (in "yyyymmdd" format) in the trace filename, specify "%d" as part of the <*filename*>.
- To include the pidnumber of the process that is generating the tracefile, specify "%p'" as part of the <*filename*>.
- To include the time (in 24-hour hhmmss format) in the trace filename, specify "%t" as part of the <*filename*>.

Examples

- Trace output goes to `/u/traces/exception.trc`. No size limit:
  ```
  -Xtrace:exception.output=/u/traces/exception.trc,maximal
  ```
- Output goes to except and wraps at 2 MB:
  ```
  -Xtrace:exception.output={except,2m},maximal
  ```
- Output filename contains today's date in yyyymmdd format (for example, `traceout.20041025.trc`):
  ```
  -Xtrace:exception.output=traceout.%d.trc,maximal
  ```
- Output file contains the number of the process (the PID number) that generated it (for example, `tracefrompid2112.trc`):
  ```
  -Xtrace:exception.output=tracefrompid%p.trc,maximal
  ```
- Output filename contains the time in hhmmss format (for example, `traceout.080312.trc`):
  ```
  -Xtrace:exception.output=traceout.%t.trc,maximal
  ```

resume:

Resumes tracing globally.

Note that suspend and resume are not recursive. That is, two suspends that are followed by a single resume cause trace to be resumed.

Example

- Trace resumed (not much use as a startup option):
  ```
  -Xtrace:resume
  ```

resumecount=<count>:

This trace option determines whether tracing is enabled for each thread.

If <*count*> is greater than zero, each thread initially has its tracing disabled and must receive <*count*> **resumethis** actions before it starts tracing.

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

This system property is for use with the **trigger** property. For more information, see "trigger=<*clause*>[,<*clause*>][,<*clause*>]..." on page 308.

Example

- Start with all tracing turned off. Each thread starts tracing when it has had three **resumethis** actions performed on it:

  ```
  -Xtrace:resumecount=3
  ```

sleeptime=nnn | aaams | bbbs:

Specify how long the sleep lasts when using the sleep trigger action.

Purpose

Use this option to determine how long a sleep trigger action lasts. The default length of time is 30 seconds. If no units are specified, the default time unit is milliseconds.

Parameters

nnn
Sleep for *nnn* milliseconds.

*aaa***ms**
Sleep for *aaa* milliseconds.

*bbb***s**
Sleep for *bbb* seconds.

stackdepth=<n>:

Used to limit the amount of stack frame information collected.

Purpose

Use this option to limit the maximum number of stack frames reported by the **jstacktrace** trace trigger action. All stack frames are recorded by default.

Parameters

n Record *n* stack frames

suspend:

Suspends tracing globally (for all threads and all forms of tracing) but leaves tracepoints activated.

Example

- Tracing suspended:

  ```
  -Xtrace:suspend
  ```

suspendcount=<count>:

This trace option determines whether tracing is enabled for each thread.

If <*count*> is greater than zero, each thread initially has its tracing enabled and must receive <*count*> suspend this action before it stops tracing.

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

This trace option is for use with the **trigger** option. For more information, see "trigger=<*clause*>[,<*clause*>][,<*clause*>]...."

Example

- Start with all tracing turned on. Each thread stops tracing when it has had three **suspendthis** actions performed on it:

```
-Xtrace:suspendcount=3
```

trigger=<clause>[,<clause>][,<clause>]...:

This trace option determines when various triggered trace actions occur. Supported actions include turning tracing on and off for all threads, turning tracing on or off for the current thread, or producing various dumps.

This trace option does not control what is traced. It controls only whether the information that has been selected by the other trace options is produced as normal or is blocked.

Each clause of the **trigger** option can be **tpnid{...}**, **method{...}**, or **group{...}**. You can specify multiple clauses of the same type if required, but you do not need to specify all types. The clause types are as follows:

method{<*methodspec*>[,<*entryAction*>[,<*exitAction*>[,<*delayCount*>[,<*matchcount*>]]]]}
: On entering a method that matches <*methodspec*>, the specified <*entryAction*> is run. On leaving a method that matches <*methodspec*>, the specified <*exitAction*> is run. If you specify a <*delayCount*>, the actions are performed only after a matching <*methodspec*> has been entered that many times. If you specify a <*matchCount*>, <*entryAction*> and <*exitAction*> are performed at most that many times.

group{<*groupname*>,<*action*>[,<*delayCount*>[,<*matchcount*>]]}
: On finding any active tracepoint that is defined as being in trace group <*groupname*>, for example **Entry** or **Exit**, the specified action is run. If you specify a <*delayCount*>, the action is performed only after that many active tracepoints from group <*groupname*> have been found. If you specify a <*matchCount*>, <*action*> is performed at most that many times.

tpnid{<*tpnid*>|<*tpnidRange*>,<*action*>[,<*delayCount*>[,<*matchcount*>]]}
: On finding the specified active <*tpnid*> (tracepoint ID) or a <*tpnid*> that falls inside the specified <*tpnidRange*>, the specified action is run. If you specify a <*delayCount*>, the action is performed only after the JVM finds such an active <*tpnid*> that many times. If you specify a <*matchCount*>, <*action*> is performed at most that many times.

Actions

Wherever an action must be specified, you must select from these choices:

abort
: Halt the JVM.

ceedump
: This action is applicable to z/OS only. For more information, see "LE CEEDUMPs" on page 226.

coredump
: See **sysdump**.

heapdump
Produce a Heapdump. See “Using Heapdump” on page 262.

javadump
Produce a Javadump. See “Using Javadump” on page 240.

jstacktrace
Examine the Java stack of the current thread and generate auxiliary tracepoints for each stack frame. The auxiliary tracepoints are written to the same destination as the tracepoint or method trace that triggered the action. You can control the number of stack frames examined with the **stackdepth=n** option. See “stackdepth=<n>” on page 307.

resume
Resume all tracing (except for threads that are suspended by the action of the **resumecount** property and Trace.suspendThis() calls).

resumethis
Decrement the suspend count for this thread. If the suspend count is zero or less, resume tracing for this thread.

segv
Cause a segmentation violation. (Intended for use in debugging.)

sleep
Delay the current thread for a length of time controlled by the **sleeptime** option. The default is 30 seconds. See “sleeptime=nnn|aaams|bbbs” on page 307.

snap
Snap all active trace buffers to a file in the current working directory. The file name has the format: `Snapnnnn.yyyymmdd.hhmmssth.ppppp.trc`, where `nnnn` is the sequence number of the snap file since JVM startup, `yyyymmdd` is the date, `hhmmssth` is the time, and `ppppp` is the process ID in decimal with leading zeros removed.

suspend
Suspend all tracing (except for special trace points).

suspendthis
Increment the suspend count for this thread. If the suspend-count is greater than zero, prevent all tracing for this thread.

sysdump (or coredump)
Produce a system dump. See “Using system dumps and the dump viewer” on page 271.

Examples

- To start tracing this thread when it enters any method in java/lang/String, and to stop tracing the thread after exiting the method:

  ```
  -Xtrace:resumecount=1
  -Xtrace:trigger=method{java/lang/String.*,resumethis,suspendthis}
  ```

- To resume all tracing when any thread enters a method in any class that starts with “error”:

  ```
  -Xtrace:trigger=method{*.error*,resume}
  ```

- To produce a core dump when you reach the 1000th and 1001st tracepoint from the “jvmri” trace group.

  Note: Without <*matchcount*>, you risk filling your disk with coredump files.

  ```
  -Xtrace:trigger=group{staticmethods,coredump,1000,2}
  ```

If using the **trigger** option generates multiple dumps in rapid succession (more than one per second), specify a dump option to guarantee unique dump names. See "Using dump agents" on page 221 for more information.

- To trace (all threads) while the application is active; that is, not starting or shut down. (The application name is "HelloWorld"):

  ```
  -Xtrace:suspend,trigger=method{HelloWorld.main,resume,suspend}
  ```

- To print a Java stack trace to the console when the `mycomponent.1` tracepoint is reached:

  ```
  -Xtrace:print=mycomponent.1,trigger=tpnid{mycomponent.1,jstacktrace}
  ```

- To write a Java stack trace to the trace output file when the Sample.code() method is called:

  ```
  -Xtrace:maximal=mt,output=trc.out,methods={mycompany/mypackage/Sample.code},
  trigger=method{mycompany/mypackage/Sample.code,jstacktrace}
  ```

what:

This trace option shows the current trace settings.

Example

```
-Xtrace:what
```

Example output:

```
Trace engine configuration
--------------------------
-Xtrace:FORMAT=C:\Java\jre\bin;C:\Java\jre\lib;.
-Xtrace:LIBPATH=C:\Java\jre\bin
-Xtrace:MAXIMAL=all{level1}
-Xtrace:EXCEPTION=j9mm{gclogger}
-Xtrace:what
--------------------------
```

Using the Java API:

You can dynamically control trace in a number of ways from a Java application by using the com.ibm.jvm.Trace class.

Activating and deactivating tracepoints

```
int  set(String cmd);
```

The Trace.set() method allows a Java application to select tracepoints dynamically. For example:

```
Trace.set("iprint=all");
```

The syntax is the same as that used in a trace properties file for the print, iprint, count, maximal, minimal and external trace options.

A single trace command is parsed per invocation of Trace.set, so to achieve the equivalent of **-Xtrace:maximal=j9mm,iprint=j9shr** two calls to Trace.set are needed with the parameters `maximal=j9mm` and `iprint=j9shr`

Obtaining snapshots of trace buffers

```
void snap();
```

You must have activated trace previously with the **maximal** or **minimal** options and without the **out** option.

Suspending or resuming trace

`void suspend();`

The `Trace.suspend()` method suspends tracing for all the threads in the JVM.

`void resume();`

The `Trace.resume()` method resumes tracing for all threads in the JVM. It is not recursive.

`void suspendThis();`

The Trace.suspendThis() method decrements the suspend and resume count for the current thread and suspends tracing the thread if the result is negative.

`void resumeThis();`

The Trace.resumeThis() method increments the suspend and resume count for the current thread and resumes tracing the thread if the result is not negative.

Using the trace formatter

The trace formatter is a Java program that converts binary trace point data in a trace file to a readable form. The formatter requires the `J9TraceFormat.dat` file, which contains the formatting templates. The formatter produces a file containing header information about the JVM that produced the binary trace file, a list of threads for which trace points were produced, and the formatted trace points with their timestamp, thread ID, trace point ID and trace point data.

To use the trace formatter on a binary trace file type:

`java com.ibm.jvm.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.

Determining the tracepoint ID of a tracepoint

Throughout the code that makes up the JVM, there are numerous tracepoints. Each tracepoint maps to a unique id consisting of the name of the component containing the tracepoint, followed by a period (".") and then the numeric identifier of the tracepoint.

These tracepoints are also recorded in two `.dat` files (`TraceFormat.dat` and `J9TraceFormat.dat`) that are shipped with the JRE, and the trace formatter uses these files to convert compressed trace points into readable form.

JVM developers and Service can use the two `.dat` files to enable formulation of trace point ids and ranges for use under **-Xtrace** when tracking down problems. The next sample is taken from the beginning of `J9TraceFormat.dat`, which illustrates how this mechanism works:

```
5.1
j9bcu.0 0 1 1 N Trc_BCU_VMInitStages_Event1 "  Trace engine initialized for module j9dyn"
j9bcu.1 2 1 1 N Trc_BCU_internalDefineClass_Entry " >internalDefineClass %p"
j9bcu.2 4 1 1 N Trc_BCU_internalDefineClass_Exit " <internalDefineClass %p ->"
j9bcu.3 2 1 1 N Trc_BCU_createRomClassEndian_Entry " >createRomClassEndian searchFilename=%s"
```

The first line of the `.dat` file is an internal version number. Following the version number is a line for each tracepoint. Trace point j9bcu.0 maps to Trc_BCU_VMInitStages_Event1 for example and j9bcu.2 maps to Trc_BCU_internalDefineClass_Exit.

The format of each tracepoint entry is:

<component.id> <t> <o> <l> <e> <symbol> <template>

where:

<component.id>
is the SDK component name.

<t> is the tracepoint type (0 through 12), where these types are used:
- 0 = event
- 1 = exception
- 2 = function entry
- 4 = function exit
- 5 = function exit with exception
- 8 = internal
- 12 = assert

<o> is the overhead (0 through 10), which determines whether the tracepoint is compiled into the runtime JVM code.

<l> is the level of the tracepoint (0 through 9). High frequency tracepoints, known as hot tracepoints, are assigned higher level numbers.

<e> is an internal flag (Y/N) and no longer used.

<*symbol*>
is the internal symbolic name of the tracepoint.
<*template*>
is a template in double quotation marks that is used to format the entry.

For example, if you discover that a problem occurred somewhere close to the issue of Trc_BCU_VMInitStages_Event, you can rerun the application with **-Xtrace:print=tpnid{j9bcu.0}**. That command will result in an output such as:

```
14:10:42.717*0x41508a00   j9bcu.0        - Trace engine initialized for module j9dyn
```

The example given is fairly trivial. However, the use of tpnid ranges and the formatted parameters contained in most trace entries provides a very powerful problem debugging mechanism.

The `.dat` files contain a list of all the tracepoints ordered by component, then sequentially numbered from 0. The full tracepoint id is included in all formatted output of a tracepoint; For example, tracing to the console or formatted binary trace.

The format of trace entries and the contents of the `.dat` files are subject to change without notice. However, the version number should guarantee a particular format.

Application trace

Application trace allows you to trace Java applications using the JVM trace facility.

You must register your Java application with application trace and add trace calls where appropriate. After you have started an application trace module, you can enable or disable individual tracepoints at any time.

Implementing application trace:

Application trace is in the package com.ibm.jvm.Trace. The application trace API is described in this section.

Registering for trace:

Use the `registerApplication()` method to specify the application to register with application trace.

The method is of the form:

```
int registerApplication(String application_name, String[] format_template)
```

The `application_name` argument is the name of the application you want to trace. The name must be the same as the application name you specify at JVM startup. The `format_template` argument is an array of format strings like the strings used by the printf method. You can specify templates of up to 16 KB. The position in the array determines the tracepoint identifier (starting at 0). You can use these identifiers to enable specific tracepoints at run time. The first character of each template is a digit that identifies the type of tracepoint. The tracepoint type can be one of entry, exit, event, exception, or exception exit. After the tracepoint type character, the template has a blank character, followed by the format string.

The trace types are defined as static values within the Trace class:

```
public static final String EVENT= "0 ";
public static final String EXCEPTION= "1 ";
public static final String ENTRY= "2 ";
public static final String EXIT= "4 ";
public static final String EXCEPTION_EXIT= "5 ";
```

The `registerApplication()` method returns an integer value. Use this value in subsequent `trace()` calls. If the `registerApplication()` method call fails for any reason, the value returned is -1.

Tracepoints:

These trace methods are implemented.

```
void trace(int handle, int traceId);
void trace(int handle, int traceId, String s1);
void trace(int handle, int traceId, String s1, String s2);
void trace(int handle, int traceId, String s1, String s2, String s3);
void trace(int handle, int traceId, String s1, Object o1);
void trace(int handle, int traceId, Object o1, String s1);
void trace(int handle, int traceId, String s1, int i1);
void trace(int handle, int traceId, int i1, String s1);
void trace(int handle, int traceId, String s1, long l1);
void trace(int handle, int traceId, long l1, String s1);
void trace(int handle, int traceId, String s1, byte b1);
void trace(int handle, int traceId, byte b1, String s1);
void trace(int handle, int traceId, String s1, char c1);
void trace(int handle, int traceId, char c1, String s1);
void trace(int handle, int traceId, String s1, float f1);
void trace(int handle, int traceId, float f1, String s1);
void trace(int handle, int traceId, String s1, double d1);
void trace(int handle, int traceId, double d1, String s1);
void trace(int handle, int traceId, Object o1);
void trace(int handle, int traceId, Object o1, Object o2);
void trace(int handle, int traceId, int i1);
void trace(int handle, int traceId, int i1, int i2);
void trace(int handle, int traceId, int i1, int i2, int i3);
void trace(int handle, int traceId, long l1);
void trace(int handle, int traceId, long l1, long l2);
void trace(int handle, int traceId, long l1, long l2, long i3);
void trace(int handle, int traceId, byte b1);
void trace(int handle, int traceId, byte b1, byte b2);
void trace(int handle, int traceId, byte b1, byte b2, byte b3);
void trace(int handle, int traceId, char c1);
void trace(int handle, int traceId, char c1, char c2);
void trace(int handle, int traceId, char c1, char c2, char c3);
void trace(int handle, int traceId, float f1);
void trace(int handle, int traceId, float f1, float f2);
void trace(int handle, int traceId, float f1, float f2, float f3);
void trace(int handle, int traceId, double d1);
void trace(int handle, int traceId, double d1, double d2);
void trace(int handle, int traceId, double d1, double d2, double d3);
void trace(int handle, int traceId, String s1, Object o1, String s2);
void trace(int handle, int traceId, Object o1, String s1, Object o2);
void trace(int handle, int traceId, String s1, int i1, String s2);
void trace(int handle, int traceId, int i1, String s1, int i2);
void trace(int handle, int traceId, String s1, long l1, String s2);
```

```
void trace(int handle, int traceId, long l1, String s1, long l2);
void trace(int handle, int traceId, String s1, byte b1, String s2);
void trace(int handle, int traceId, byte b1, String s1, byte b2);
void trace(int handle, int traceId, String s1, char c1, String s2);
void trace(int handle, int traceId, char c1, String s1, char c2);
void trace(int handle, int traceId, String s1, float f1, String s2);
void trace(int handle, int traceId, float f1, String s1, float f2);
void trace(int handle, int traceId, String s1, double d1, String s2);
void trace(int handle, int traceId, double d1, String s1, double d2);
```

The handle argument is the value returned by the registerApplication() method. The traceId argument is the number of the template entry starting at 0.

Printf specifiers:

Application trace supports the ANSI C printf specifiers. You must be careful when you select the specifier; otherwise you might get unpredictable results, including abnormal termination of the JVM.

For 64-bit integers, you must use the ll (lowercase LL, meaning long long) modifier. For example: `%lld` or `%lli`.

For pointer-sized integers use the z modifier. For example: `%zx` or `%zd`.

Example HelloWorld with application trace:

This code illustrates a "HelloWorld" application with application trace.

For more information about this example, see "Using application trace at run time" on page 316.

```
import com.ibm.jvm.Trace;
public class HelloWorld
{
   static int handle;
   static String[] templates;
   public static void main ( String[] args )
   {
      templates = new String[ 5 ];
      templates[ 0 ] = Trace.ENTRY          + "Entering %s";
      templates[ 1 ] = Trace.EXIT           + "Exiting %s";
      templates[ 2 ] = Trace.EVENT          + "Event id %d, text = %s";
      templates[ 3 ] = Trace.EXCEPTION      + "Exception: %s";
      templates[ 4 ] = Trace.EXCEPTION_EXIT + "Exception exit from %s";

      // Register a trace application called HelloWorld
      handle = Trace.registerApplication( "HelloWorld", templates );

      // Set any tracepoints that are requested on the command line
      for ( int i = 0; i < args.length; i++ )
      {
         System.err.println( "Trace setting: " + args[ i ] );
         Trace.set( args[ i ] );
      }

      // Trace something....
      Trace.trace( handle, 2, 1, "Trace initialized" );

      // Call a few methods...
      sayHello( );
      sayGoodbye( );
   }
```

```
   private static void sayHello( )
   {
      Trace.trace( handle, 0, "sayHello" );
      System.out.println( "Hello" );
      Trace.trace( handle, 1, "sayHello" );
   }

   private static void sayGoodbye( )
   {
      Trace.trace( handle, 0, "sayGoodbye" );
      System.out.println( "Bye" );
      Trace.trace( handle, 4, "sayGoodbye" );
   }
}
```

Using application trace at run time:

At run time, you can enable one or more applications for application trace.

The "Example HelloWorld with application trace" on page 315 uses the Trace.set() API to pass arguments to the trace function. For example, to pass the **iprint** argument to the trace function, use the following command:

```
java HelloWorld iprint=HelloWorld
```

Starting the example **HelloWorld** application in this way produces the following results:

```
Trace setting: iprint=HelloWorld
09:50:29.417*0x2a08a00 084002   - Event id 1, text = Trace initialized
09:50:29.417 0x2a08a00 084000   > Entering sayHello
Hello
09:50:29.427 0x2a08a00 084001   < Exiting sayHello
09:50:29.427 0x2a08a00 084000   > Entering sayGoodbye
Bye
09:50:29.437 0x2a08a00 084004 * < Exception exit from sayGoodbye
```

You can also specify trace options directly by using the **-Xtrace** option. See "Options that control tracepoint activation" on page 297 for more details. For example, you can obtain a similar result to the previous command by using the **-Xtrace** option to specify **iprint** on the command line:

```
java -Xtrace:iprint=HelloWorld HelloWorld
```

Note: You can enable tracepoints by application name and by tracepoint number. Using tracepoint "levels" or "types" is not supported for application trace.

Using method trace

Method trace is a powerful tool for tracing methods in any Java code.

Method trace provides a comprehensive and detailed diagnosis of code paths inside your application, and also inside the system classes. You do not have to add any hooks or calls to existing code. You can focus on interesting code by using wild cards and filtering to control method trace.

Method trace can trace:

- Method entry
- Method exit

Use method trace to debug and trace application code and the system classes provided with the JVM.

While method trace is powerful, it also has a cost. Application throughput is affected by method trace. Additionally, trace output is reasonably large and might require a large amount of drive space. For instance, a full method trace of a “Hello World” application is over 10 MB.

Running with method trace:

Control method trace by using the command-line option **-Xtrace:**<*option*>.

To produce method trace you need to set trace options for the Java classes and methods you want to trace. You also need to route the method trace to the destination you require.

You must set the following two options:

1. Use **-Xtrace:methods** to select which Java classes and methods you want to trace. You can use IBM Monitoring and Diagnostic Tools for Java - Health Center to monitor your application to see which methods should be traced. You can also use the tool to generate the required **-Xtrace** parameters, and view the resulting data.
2. Use either
   - **-Xtrace:print** to route the trace to stderr.
   - **-Xtrace:maximal** and **-Xtrace:output** to route the trace to a binary compressed file using memory buffers.

Use the **methods** parameter to control what is traced. For example, to trace all methods on the String class, set **-Xtrace:methods=java/lang/String.*,print=mt**.

The **methods** parameter is formally defined as follows:

```
-Xtrace:methods=[[!]<method_spec>[,...]]
```

Where *<method_spec>* is formally defined as:

```
{*|[*]<classname>[*]}.{*|[*]<methodname>[*]}[()]
```

Notes:

- The exclamation point (!) in the **methods** parameter is a NOT operator. You can use this symbol and multiple methods in combination. For example, the following option traces all methods in the java.util.HashMap class except those beginning with `put`:

  ```
  -Xtrace:methods={java/util/HashMap.*,!java/util/HashMap.put*},print=mt
  ```

- The parentheses, (), that are in the *<method_spec>* variable define whether to trace method parameters. Method call parameters are traced only for interpreted methods. If the method was compiled by the JIT compiler, the parameters are not traced.
- If a method specification includes commas, the whole specification must be enclosed in braces:

  ```
  -Xtrace:methods={java/lang/*,java/util/*},print=mt
  ```

- You might have to enclose your command line in quotation marks. This action prevents the shell intercepting and fragmenting comma-separated command lines:

  ```
  "-Xtrace:methods={java/lang/*,java/util/*},print=mt"
  ```

Use the **print**, **maximal** and **output** options to route the trace to the required destination, where:

- **print** formats the trace point data while the Java application is running and writes the tracepoints to stderr.
- **maximal** saves the trace points into memory buffers.
- **output** writes the memory buffers to a file, in a binary compressed format.

To produce method trace that is routed to stderr, use the **print** option, specifying **mt** (method trace). For example: **-Xtrace:methods=java/lang/String.*,print=mt**.

To produce method trace that is written to a binary file from the memory buffers, use the **maximal** and **output** options. For example: **-Xtrace:methods=java/lang/String.*,maximal=mt,output=mytrace.trc**.

If you want your trace output to contain only the tracepoints you specify, use the option **-Xtrace:none** to switch off the default tracepoints. For example: **java -Xtrace:none -Xtrace:methods=java/lang/String.*,maximal=mt,output=mytrace.trc <class>**.

Untraceable methods:

Internal Native Library (INL) native methods inside the JVM cannot be traced because they are not implemented using JNI. The list of methods that are not traceable is subject to change without notice between releases.

The INL native methods in the JVM include:

```
java.lang.Class.allocateAndFillArray
java.lang.Class.forNameImpl
java.lang.Class.getClassDepth
java.lang.Class.getClassLoaderImpl
java.lang.Class.getComponentType
java.lang.Class.getConstructorImpl
java.lang.Class.getConstructorsImpl
java.lang.Class.getDeclaredClassesImpl
java.lang.Class.getDeclaredConstructorImpl
java.lang.Class.getDeclaredConstructorsImpl
java.lang.Class.getDeclaredFieldImpl
java.lang.Class.getDeclaredFieldsImpl
java.lang.Class.getDeclaredMethodImpl
java.lang.Class.getDeclaredMethodsImpl
java.lang.Class.getDeclaringClassImpl
java.lang.Class.getEnclosingObject
java.lang.Class.getEnclosingObjectClass
java.lang.Class.getFieldImpl
java.lang.Class.getFieldsImpl
java.lang.Class.getGenericSignature
java.lang.Class.getInterfaceMethodCountImpl
java.lang.Class.getInterfaceMethodsImpl
java.lang.Class.getInterfaces
java.lang.Class.getMethodImpl
java.lang.Class.getModifiersImpl
java.lang.Class.getNameImpl
java.lang.Class.getSimpleNameImpl
java.lang.Class.getStackClass
java.lang.Class.getStackClasses
java.lang.Class.getStaticMethodCountImpl
java.lang.Class.getStaticMethodsImpl
java.lang.Class.getSuperclass
java.lang.Class.getVirtualMethodCountImpl
java.lang.Class.getVirtualMethodsImpl
java.lang.Class.isArray
java.lang.Class.isAssignableFrom
java.lang.Class.isInstance
java.lang.Class.isPrimitive
```

```
java.lang.Class.newInstanceImpl
java.lang.ClassLoader.findLoadedClassImpl
java.lang.ClassLoader.getStackClassLoader
java.lang.ClassLoader.loadLibraryWithPath
java.lang.J9VMInternals.getInitStatus
java.lang.J9VMInternals.getInitThread
java.lang.J9VMInternals.initializeImpl
java.lang.J9VMInternals.sendClassPrepareEvent
java.lang.J9VMInternals.setInitStatusImpl
java.lang.J9VMInternals.setInitThread
java.lang.J9VMInternals.verifyImpl
java.lang.J9VMInternals.getStackTrace
java.lang.Object.clone
java.lang.Object.getClass
java.lang.Object.hashCode
java.lang.Object.notify
java.lang.Object.notifyAll
java.lang.Object.wait
java.lang.ref.Finalizer.runAllFinalizersImpl
java.lang.ref.Finalizer.runFinalizationImpl
java.lang.ref.Reference.getImpl
java.lang.ref.Reference.initReferenceImpl
java.lang.reflect.AccessibleObject.checkAccessibility
java.lang.reflect.AccessibleObject.getAccessibleImpl
java.lang.reflect.AccessibleObject.getExceptionTypesImpl
java.lang.reflect.AccessibleObject.getModifiersImpl
java.lang.reflect.AccessibleObject.getParameterTypesImpl
java.lang.reflect.AccessibleObject.getSignature
java.lang.reflect.AccessibleObject.getStackClass
java.lang.reflect.AccessibleObject.initializeClass
java.lang.reflect.AccessibleObject.invokeImpl
java.lang.reflect.AccessibleObject.setAccessibleImpl
java.lang.reflect.Array.get
java.lang.reflect.Array.getBoolean
java.lang.reflect.Array.getByte
java.lang.reflect.Array.getChar
java.lang.reflect.Array.getDouble
java.lang.reflect.Array.getFloat
java.lang.reflect.Array.getInt
java.lang.reflect.Array.getLength
java.lang.reflect.Array.getLong
java.lang.reflect.Array.getShort
java.lang.reflect.Array.multiNewArrayImpl
java.lang.reflect.Array.newArrayImpl
java.lang.reflect.Array.set
java.lang.reflect.Array.setBoolean
java.lang.reflect.Array.setByte
java.lang.reflect.Array.setChar
java.lang.reflect.Array.setDouble
java.lang.reflect.Array.setFloat
java.lang.reflect.Array.setImpl
java.lang.reflect.Array.setInt
java.lang.reflect.Array.setLong
java.lang.reflect.Array.setShort
java.lang.reflect.Constructor.newInstanceImpl
java.lang.reflect.Field.getBooleanImpl
java.lang.reflect.Field.getByteImpl
java.lang.reflect.Field.getCharImpl
java.lang.reflect.Field.getDoubleImpl
java.lang.reflect.Field.getFloatImpl
java.lang.reflect.Field.getImpl
java.lang.reflect.Field.getIntImpl
java.lang.reflect.Field.getLongImpl
java.lang.reflect.Field.getModifiersImpl
java.lang.reflect.Field.getNameImpl
java.lang.reflect.Field.getShortImpl
java.lang.reflect.Field.getSignature
```

```
java.lang.reflect.Field.getTypeImpl
java.lang.reflect.Field.setBooleanImpl
java.lang.reflect.Field.setByteImpl
java.lang.reflect.Field.setCharImpl
java.lang.reflect.Field.setDoubleImpl
java.lang.reflect.Field.setFloatImpl
java.lang.reflect.Field.setImpl
java.lang.reflect.Field.setIntImpl
java.lang.reflect.Field.setLongImpl
java.lang.reflect.Field.setShortImpl
java.lang.reflect.Method.getNameImpl
java.lang.reflect.Method.getReturnTypeImpl
java.lang.String.intern
java.lang.String.isResettableJVM0
java.lang.System.arraycopy
java.lang.System.currentTimeMillis
java.lang.System.hiresClockImpl
java.lang.System.hiresFrequencyImpl
java.lang.System.identityHashCode
java.lang.System.nanoTime
java.lang.Thread.currentThread
java.lang.Thread.getStackTraceImpl
java.lang.Thread.holdsLock
java.lang.Thread.interrupted
java.lang.Thread.interruptImpl
java.lang.Thread.isInterruptedImpl
java.lang.Thread.resumeImpl
java.lang.Thread.sleep
java.lang.Thread.startImpl
java.lang.Thread.stopImpl
java.lang.Thread.suspendImpl
java.lang.Thread.yield
java.lang.Throwable.fillInStackTrace
java.security.AccessController.getAccessControlContext
java.security.AccessController.getProtectionDomains
java.security.AccessController.getProtectionDomainsImpl
org.apache.harmony.kernel.vm.VM.getStackClassLoader
org.apache.harmony.kernel.vm.VM.internImpl
```

Examples of use:

Here are some examples of method trace commands and their results.

- **Tracing entry and exit of all methods in a given class:**

  ```
  -Xtrace:methods=java/lang/String.*,print=mt
  ```

  This example traces entry and exit of all methods in the java.lang.String class. The name of the class must include the full package name, using '/' as a separator. The method name is separated from the class name by a dot '.' In this example, '*' is used to include all methods. Sample output:

  ```
  09:39:05.569 0x1a1100 mt.0 > java/lang/String.length()I Bytecode method,
  This = 8b27d8
  09:39:05.579 0x1a1100 mt.6 < java/lang/String.length()I Bytecode method
  ```

- **Tracing method input parameters:**

  ```
  -Xtrace:methods=java/lang/Thread.*(),print=mt
  ```

  This example traces all methods in the java.lang.Thread class, with the parentheses '()' indicating that the trace should also include the method call parameters. The output includes an extra line, giving the class and location of the object on which the method was called, and the values of the parameters. In this example the method call is Thread.join(long millis,int nanos), which has two parameters:

```
09:58:12.949 0x4236ce00 mt.0 > java/lang/Thread.join(JI)V Bytecode method, This = 8ffd20
09:58:12.959 0x4236ce00 mt.18 - Instance method receiver: com/ibm/tools/attach/javaSE/AttachHandler@008FFD20
arguments: ((long)1000,(int)0)
```

Method call parameters are traced only for interpreted methods. If the method was compiled by the JIT compiler, the parameters are not provided. For example:

```
16:56:45.636 0x3e70000 mt.0 > java/lang/Thread.join(JI)V Bytecode method, This = 7ff7e7ca450
16:56:45.648 0x3e70000 mt.18 - Instance method receiver: com/ibm/tools/attach/javaSE/AttachHandler@000007FF7E7CA450
arguments: ((long)10000,(int)0)
16:56:55.726*0x3e70000 mt.6 < java/lang/Thread.join(JI)V Bytecode method
16:56:56.462 0x3e70000 mt.1 > java/lang/Thread.join(JI)V Compiled method, This = 7ff7e7ca450
16:56:56.617 0x3e70000 mt.7 < java/lang/Thread.join(JI)V Compiled method
```

- **Tracing multiple methods:**

  ```
  -Xtrace:methods={java/util/HashMap.size,java/lang/String.length},print=mt
  ```

  This example traces the size method on the java.util.HashMap class and the length method on the java.lang.String class. The method specification includes the two methods separated by a comma, with the entire method specification enclosed in braces '{' and '}'. Sample output:

  ```
  10:28:19.296 0x1a1100 mt.0 > java/lang/String.length()I Bytecode method,
  This = 8c2548
  10:28:19.306 0x1a1100 mt.6 < java/lang/String.length()I Bytecode method
  10:28:19.316 0x1a1100 mt.0 > java/util/HashMap.size()I Bytecode method,
  This = 8dd7e8
  10:28:19.326 0x1a1100 mt.6 < java/util/HashMap.size()I Bytecode method
  ```

- **Using the ! (not) operator to select tracepoints:**

  ```
  -Xtrace:methods={java/util/HashMap.*,!java/util/HashMap.put*},print
  ```

  This example traces all methods in the java.util.HashMap class except those beginning with `put`. Sample output:

```
10:37:42.225 0x1a1100 mt.0 > java/util/HashMap.createHashedEntry(Ljava/lang/Object;II)Ljava/util/
HashMap$Entry; Bytecode method, This = 8e09e0
10:37:42.246 0x1a1100 mt.6 < java/util/HashMap.createHashedEntry(Ljava/lang/Object;II)Ljava/util/
HashMap$Entry; Bytecode method
10:37:42.256 0x1a1100 mt.1 > java/util/HashMap.findNonNullKeyEntry(Ljava/lang/Object;II)Ljava/util/
HashMap$Entry; Compiled method, This = 8dd7e0
10:37:42.266 0x1a1100 mt.7 < java/util/HashMap.findNonNullKeyEntry(Ljava/lang/Object;II)Ljava/util/
HashMap$Entry; Compiled method
```

Example of method trace output:

An example of method trace output.

Sample output using the command **java -Xtrace:iprint=mt,methods=java/lang/*.* -version**:

```
10:02:42.281*0x9e900     mt.4         > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                             V Compiled static method
10:02:42.281 0x9e900     mt.4          > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                             V Compiled static method
10:02:42.281 0x9e900     mt.4           > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                             V Compiled static method
10:02:42.281 0x9e900     mt.4            > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                             V Compiled static method
10:02:42.281 0x9e900     mt.10           < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                             V Compiled static method
10:02:42.281 0x9e900     mt.10          < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                             V Compiled static method
10:02:42.281 0x9e900     mt.4           > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                             V Compiled static method
10:02:42.281 0x9e900     mt.10          < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
```

```
                    V Compiled static method
10:02:42.281 0x9e900      mt.10         < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                    V Compiled static method
10:02:42.281 0x9e900      mt.4          > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                    V Compiled static method
10:02:42.281 0x9e900      mt.4           > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                    V Compiled static method
10:02:42.296 0x9e900      mt.10          < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                    V Compiled static method
10:02:42.296 0x9e900      mt.10         < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                    V Compiled static method
10:02:42.296 0x9e900      mt.4          > java/lang/String.<clinit>()V Compiled static method
10:02:42.296 0x9e900      mt.4           > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                    V Compiled static method
10:02:42.296 0x9e900      mt.4            > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                    V Compiled static method
10:02:42.296 0x9e900      mt.4             > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                    V Compiled static method
10:02:42.296 0x9e900      mt.10            < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                    V Compiled static method
10:02:42.296 0x9e900      mt.4             > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                    V Compiled static method
10:02:42.328 0x9e900      mt.10            < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                    V Compiled static method
10:02:42.328 0x9e900      mt.10           < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                    V Compiled static method
10:02:42.328 0x9e900      mt.4            > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                    V Compiled static method
10:02:42.328 0x9e900      mt.10           < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                    V Compiled static method
10:02:42.328 0x9e900      mt.4            > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                    V Compiled static method
10:02:42.328 0x9e900      mt.10           < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                    V Compiled static method
10:02:42.328 0x9e900      mt.10          < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                    V Compiled static method
```

The output lines comprise:

- `0x9e900`, the current **execenv** (execution environment). Because every JVM thread has its own **execenv**, you can regard **execenv** as a thread-id. All trace with the same **execenv** relates to a single thread.
- The individual tracepoint id in the mt component that collects and emits the data.
- The remaining fields show whether a method is being entered (>) or exited (<), followed by details of the method.

JIT and AOT problem determination

You can use command-line options to help diagnose JIT and AOT compiler problems and to tune performance.

- "Diagnosing a JIT or AOT problem"
- "Performance of short-running applications" on page 328
- "JVM behavior during idle periods" on page 328

Diagnosing a JIT or AOT problem

Occasionally, valid bytecodes might compile into invalid native code, causing the Java program to fail. By determining whether the JIT or AOT compiler is faulty and, if so, *where* it is faulty, you can provide valuable help to the Java service team.

About this task

This section describes how you can determine if your problem is compiler-related. This section also suggests some possible workarounds and debugging techniques for solving compiler-related problems.

Disabling the JIT or AOT compiler:

If you suspect that a problem is occurring in the JIT or AOT compiler, disable compilation to see if the problem remains. If the problem still occurs, you know that the compiler is not the cause of it.

About this task

The JIT compiler is enabled by default. The AOT compiler is also enabled, but, is not active unless shared classes have been enabled. For efficiency reasons, not all methods in a Java application are compiled. The JVM maintains a call count for each method in the application; every time a method is called and interpreted, the call count for that method is incremented. When the count reaches the compilation threshold, the method is compiled and executed natively.

The call count mechanism spreads compilation of methods throughout the life of an application, giving higher priority to methods that are used most frequently. Some infrequently used methods might never be compiled at all. As a result, when a Java program fails, the problem might be in the JIT or AOT compiler or it might be elsewhere in the JVM.

The first step in diagnosing the failure is to determine *where* the problem is. To do this, you must first run your Java program in purely interpreted mode (that is, with the JIT and AOT compilers disabled).

Procedure

1. Remove any **-Xjit** and **-Xaot** options (and accompanying parameters) from your command line.
2. Use the **-Xint** command-line option to disable the JIT and AOT compilers. For performance reasons, do not use the **-Xint** option in a production environment.

What to do next

Running the Java program with the compilation disabled leads to one of the following situations:

- The failure remains. The problem is not in the JIT or AOT compiler. In some cases, the program might start failing in a different manner; nevertheless, the problem is not related to the compiler.
- The failure disappears. The problem is most likely in the JIT or AOT compiler.

  If you are not using shared classes, the JIT compiler is at fault. If you are using shared classes, you must determine which compiler is at fault by running your

application with only JIT compilation enabled. Run your application with the **-Xnoaot** option instead of the **-Xint** option. This leads to one of the following situations:

- The failure remains. The problem is in the JIT compiler. You can also use the **-Xnojit** instead of the **-Xnoaot** option to ensure that only the JIT compiler is at fault.
- The failure disappears. The problem is in the AOT compiler.

Selectively disabling the JIT or AOT compiler:

If your Java program failure points to a problem with the JIT or AOT compiler, you can try to narrow down the problem further.

About this task

By default, the JIT compiler optimizes methods at various optimization levels. Different selections of optimizations are applied to different methods, which are based on their call counts. Methods that are called more frequently are optimized at higher levels. By changing JIT compiler parameters, you can control the optimization level at which methods are optimized. You can determine whether the optimizer is at fault and, if it is, which optimization is problematic.

In contrast, the AOT compiler compiles methods only at the "cold" optimization level. Forcing the AOT compiler to compile a method at a higher level is not supported.

You specify JIT parameters as a comma-separated list, which is appended to the **-Xjit** option. The syntax is **-Xjit**:<*param1*>,<*param2*>=<*value*>. For example:

```
java -Xjit:verbose,optLevel=noOpt HelloWorld
```

runs the HelloWorld program, enables verbose output from the JIT, and makes the JIT generate native code without performing any optimizations. Optimization options are listed in "How the JIT compiler optimizes code" on page 58. The AOT compiler is controlled in a similar manner, by using the **-Xaot** option. Use the **-Xjit** option when you are diagnosing JIT compiler problems, and the **-Xaot** option when you are diagnosing AOT compiler problems.

Follow these steps to determine which part of the compiler is causing the failure:

Procedure

1. Set the JIT or AOT parameter **count=0** to change the compilation threshold to zero. This parameter causes each Java method to be compiled before it is run. Use **count=0** only when you are diagnosing problems, because a lot more methods are compiled, including methods that are used infrequently. The extra compilation uses more computing resources and slows down your application. With **count=0**, your application fails immediately when the problem area is reached. In some cases, by using **count=1** can reproduce the failure more reliably.
2. Add **disableInlining** to the JIT or AOT compiler parameters. **disableInlining** disables the generation of larger and more complex code. If the problem no longer occurs, use **disableInlining** as a workaround while the Java service team analyzes and fixes the compiler problem.
3. Decrease the optimization levels by adding the **optLevel** parameter, and run the program again until the failure no longer occurs, or you reach the "noOpt" level. For a JIT compiler problem, start with "scorching" and work down the

list. For an AOT compiler problem, start with "cold" and work down the list. The optimization levels are, in decreasing order:

a. scorching
b. veryHot
c. hot
d. warm
e. cold
f. noOpt

What to do next

If one of these settings causes your failure to disappear, you have a workaround that you can use. This workaround is temporary while the Java service team analyze and fix the compiler problem. If removing **disableInlining** from the JIT or AOT parameter list does not cause the failure to reappear, do so to improve performance. Follow the instructions in "Locating the failing method" to improve the performance of the workaround.

If the failure still occurs at the "noOpt" optimization level, you must disable the JIT or AOT compiler as a workaround.

Locating the failing method:

When you have determined the lowest optimization level at which the JIT or AOT compiler must compile methods to trigger the failure, you can find out which part of the Java program, when compiled, causes the failure. You can then instruct the compiler to limit the workaround to a specific method, class, or package, allowing the compiler to compile the rest of the program as usual. For JIT compiler failures, if the failure occurs with **-Xjit:optLevel=noOpt**, you can also instruct the compiler to not compile the method or methods that are causing the failure at all.

Before you begin

If you see error output like this example, you can use it to identify the failing method:

```
Unhandled exception
Type=Segmentation error vmState=0x00000000
Target=2_30_20050520_01866_BHdSMr (Linux 2.4.21-27.0.2.EL)
CPU=s390x (2 logical CPUs) (0x7b6a8000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=4148bf20 Signal_Code=00000001
Handler1=00000100002ADB14 Handler2=00000100002F480C InaccessibleAddress=0000000000000000
gpr0=0000000000000006 gpr1=0000000000000006 gpr2=0000000000000000 gpr3=0000000000000006
gpr4=0000000000000001 gpr5=0000000080056808 gpr6=0000010002BCCA20 gpr7=0000000000000000
......
Compiled_method=java/security/AccessController.toArrayOfProtectionDomains([Ljava/lang/Object;
Ljava/security/AccessControlContext;)[Ljava/security/ProtectionDomain;
```

The important lines are:

vmState=0x00000000
Indicates that the code that failed was not JVM runtime code.

Module= or Module_base_address=
Not in the output (might be blank or zero) because the code was compiled by the JIT, and outside any DLL or library.

Compiled_method=
Indicates the Java method for which the compiled code was produced.

About this task

If your output does not indicate the failing method, follow these steps to identify the failing method:

Procedure

1. Run the Java program with the JIT parameters **verbose** and **vlog=**<*filename*> added to the **-Xjit** or **-Xaot** option. With these parameters, the compiler lists compiled methods in a log file named <*filename*>.<*date*>.<*time*>.<*pid*>, also called a *limit file*. A typical limit file contains lines that correspond to compiled methods, like:

   ```
   + (hot) java/lang/Math.max(II)I @ 0x10C11DA4-0x10C11DDD
   ```

   Lines that do not start with the plus sign are ignored by the compiler in the following steps and you can remove them from the file. Methods compiled by the AOT compiler start with `+ (AOT cold)`. Methods for which AOT code is loaded from the shared class cache start with `+ (AOT load)`.
2. Run the program again with the JIT or AOT parameter **limitFile=**(<*filename*>,<*m*>,<*n*>), where <*filename*> is the path to the limit file, and <*m*> and <*n*> are line numbers indicating the first and the last methods in the limit file that should be compiled. The compiler compiles only the methods listed on lines <*m*> to <*n*> in the limit file. Methods not listed in the limit file and methods listed on lines outside the range are not compiled and no AOT code in the shared data cache for those methods will be loaded. If the program no longer fails, one or more of the methods that you have removed in the last iteration must have been the cause of the failure.
3. Optional: If you are diagnosing an AOT problem, run the program a second time with the same options to allow compiled methods to be loaded from the shared data cache. You can also add the **–Xaot:scount=0** option to ensure that AOT-compiled methods stored in the shared data cache will be used when the method is first called. Some AOT compilation failures happen only when AOT-compiled code is loaded from the shared data cache. To help diagnose these problems, use the **–Xaot:scount=0** option to ensure that AOT-compiled methods stored in the shared data cache are used when the method is first called, which might make the problem easier to reproduce. Please note that if you set the **scount** option to 0 it will force AOT code loading and will pause any application thread waiting to execute that method. Thus, this should only be used for diagnostic purposes. More significant pause times can occur with the **–Xaot:scount=0** option.
4. Repeat this process using different values for <*m*> and <*n*>, as many times as necessary, to find the minimum set of methods that must be compiled to trigger the failure. By halving the number of selected lines each time, you can perform a binary search for the failing method. Often, you can reduce the file to a single line.

What to do next

When you have located the failing method, you can disable the JIT or AOT compiler for the failing method only. For example, if the method java/lang/Math.max(II)I causes the program to fail when JIT-compiled with **optLevel=hot**, you can run the program with:

```
-Xjit:{java/lang/Math.max(II)I}(optLevel=warm,count=0)
```

to compile only the failing method at an optimization level of “warm”, but compile all other methods as usual.

If a method fails when it is JIT-compiled at "noOpt", you can exclude it from compilation altogether, using the **exclude**={<*method*>} parameter:

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

If a method causes the program to fail when AOT code is compiled or loaded from the shared data cache, exclude the method from AOT compilation and AOT loading using the **exclude**={<*method*>} parameter:

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

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

Identifying JIT compilation failures:

For JIT compiler failures, analyze the error output to determine if a failure occurs when the JIT compiler attempts to compile a method.

If the JVM crashes, and you can see that the failure has occurred in the JIT library (`libj9jit<vm_version>.so` or `libj9jit25.so`), the JIT compiler might have failed during an attempt to compile a method.

If you see error output like this example, you can use it to identify the failing method:

```
Unhandled exception
Type=Segmentation error vmState=0x00050000
Target=2_30_20051215_04381_BHdSMr (Linux 2.4.21-32.0.1.EL)
CPU=ppc64 (4 logical CPUs) (0xebf4e000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=00000000 Signal_Code=00000001
Handler1=0000007FE05645B8 Handler2=0000007FE0615C20
R0=E8D4001870C00001 R1=0000007FF49181E0 R2=0000007FE2FBCEE0 R3=0000007FF4E60D70
R4=E8D4001870C00000 R5=0000007FE2E02D30 R6=0000007FF4C0F188 R7=0000007FE2F8C290
......
Module=/home/test/sdk/jre/bin/libj9jit<vm_version>.so
Module_base_address=0000007FE29A6000
......
Method_being_compiled=com/sun/tools/javac/comp/Attr.visitMethodDef(Lcom/sun/tools/javac/tree/
JCTree$JCMethodDecl;)
```

The important lines are:

vmState=0x00050000
: Indicates that the JIT compiler is compiling code. For a list of vmState code numbers, see the table in Javadump "TITLE, GPINFO, and ENVINFO sections" on page 243

Module=/home/test/sdk/jre/bin/libj9jit<*vm_version*>.so
: Indicates that the error occurred in `libj9jit<vm_version>.so`, the JIT compiler module.

Method_being_compiled=
: Indicates the Java method being compiled.

If your output does not indicate the failing method, use the **verbose** option with the following additional settings:

```
-Xjit:verbose={compileStart|compileEnd}
```

These **verbose** settings report when the JIT starts to compile a method, and when it ends. If the JIT fails on a particular method (that is, it starts compiling, but crashes before it can end), use the **exclude** parameter to exclude it from compilation (refer

to “Locating the failing method” on page 325). If excluding the method prevents the crash, you have a workaround that you can use while the service team corrects your problem.

Identifying AOT compilation failures:

AOT problem determination is very similar to JIT problem determination.

About this task

As with the JIT, first run your application with **-Xnoaot**, which ensures that the AOT'ed code is not used when running the application. If this fixes the problem, use the same technique described in “Locating the failing method” on page 325, providing the **-Xaot** option in place of the **-Xjit** option where appropriate.

Performance of short-running applications

The IBM JIT compiler is tuned for long-running applications typically used on a server. You can use the **-Xquickstart** command-line option to improve the performance of short-running applications, especially for applications in which processing is not concentrated into a few methods.

-Xquickstart causes the JIT compiler to use a lower optimization level by default and to compile fewer methods. Performing fewer compilations more quickly can improve application startup time. When the AOT compiler is active (both shared classes and AOT compilation enabled), **-Xquickstart** causes all methods selected for compilation to be AOT compiled, which improves the startup time of subsequent runs. **-Xquickstart** might degrade performance if it is used with long-running applications that contain methods using a large amount of processing resource. The implementation of **-Xquickstart** is subject to change in future releases.

You can also try improving startup times by adjusting the JIT threshold (using trial and error). See “Selectively disabling the JIT or AOT compiler” on page 324 for more information.

JVM behavior during idle periods

You can reduce the CPU cycles consumed by an idle JVM by using the **-XsamplingExpirationTime** option to turn off the JIT sampling thread.

The JIT sampling thread profiles the running Java application to discover commonly used methods. The memory and processor usage of the sampling thread is negligible, and the frequency of profiling is automatically reduced when the JVM is idle.

In some circumstances, you might want no CPU cycles consumed by an idle JVM. To do so, specify the **-XsamplingExpirationTime**<*time*> option. Set <*time*> to the number of seconds for which you want the sampling thread to run. Use this option with care; after it is turned off, you cannot reactivate the sampling thread. Allow the sampling thread to run for long enough to identify important optimizations.

The Diagnostics Collector

The Diagnostics Collector gathers the Java diagnostic files for a problem event.

Introduction to the Diagnostics Collector

The Diagnostics Collector gathers the Java diagnostic files for a problem event.

The Java runtime environment produces multiple diagnostic files in response to events such as General Protection Faults, out of memory conditions or receiving unexpected operating system signals. The Diagnostics Collector runs just after the Java runtime environment produces diagnostic files. It searches for system dumps, Java dumps, heap dumps, Java trace dumps and the verbose GC log that match the time stamp for the problem event. If a system dump is found, then optionally the Diagnostics Collector can run the **jextract** command to post-process the dump and capture extra information required to analyze system dumps. The Diagnostics Collector then produces a single .zip file containing all the diagnostic information for the problem event. Steps in the collection of diagnostic data are logged in a text file. At the end of the collection process, the log file is copied into the output .zip file.

Note: The Diagnostic Collector creates a separate process for collecting the data and producing the .zip file. This process produces console output, including messages, that is written to the same console as the original Java application. Some of these messages might therefore appear in the console after the original Java process finishes and the command prompt is displayed.

The Diagnostics Collector also has a feature to give warnings if there are JVM settings in place that could prevent the JVM from producing diagnostic data. These warnings are produced at JVM start, so that the JVM can be restarted with fixed settings if necessary. The warnings are printed on `stderr` and in the Diagnostics Collector log file. Fix the settings identified by any warning messages before restarting your Java application. Fixing warnings makes it more likely that the correct data is available for IBM Support to diagnose a Java problem.

Using the Diagnostics Collector

You can start the Diagnostics Collector after a dump has occurred to collect the relevant files.

The Diagnostics Collector is off by default and is enabled by a JVM command-line option:

```
-Xdiagnosticscollector[:settings=<filename>]
```

Specifying a Diagnostics Collector settings file is optional. By default, the settings file `jre/lib/dc.properties` is used. See “Diagnostics Collector settings” on page 332 for details of the settings available.

If you run a Java program from the command line with the Diagnostics Collector enabled, it produces some console output. The Diagnostics Collector runs asynchronously, in a separate process to the one that runs your Java program. The effect is that output appears after the command-line prompt returns from running your program. If this happens, it does not mean that the Diagnostics Collector has hung. Press `enter` to get the command-line prompt back.

If the Diagnostics Collector was not enabled when your dump occurred, you can manually run the Diagnostics Collector afterward. Use the following command:

```
java com.ibm.java.diagnostics.collector.DiagnosticsCollector [OPTIONS]
```

where`[OPTIONS]` are:

-stamp <YYYYMMDD.hhmmss.pid>
The Diagnostics Collector collects the relevant files for the dumps with the specified stamp. The following directories are searched:

- The path provided by the last dump parameter, if set.

- The parent directory provided by the last dump parameter, if set.
- The current working directory.
- The location set in the environment variables IBM_JAVACOREDIR, IBM_HEAPDUMPDIR, IBM_COREDIR, _CEE_DMPTARG, JAVA_DUMP_TDUMP_PATTERN, and TMPDIR.
- The temporary directory, which is `/tmp` (see J9_TMP_DUMP_NAME)..

-date <YYYYMMDD>
This option finds the newest dump with the specified date in the current directory. If an appropriate dump is found, the Diagnostics Collector looks for the files to collect in the directories listed for the **-stamp** option.

-lastdump <path>
Where **<path>** is the full path for the last dump. This location is the first directory that the Diagnostics Collector searches for relevant dump files.

If no options are set, the Diagnostics Collector attempts to find the newest dump in the current directory. If a dump is found, the Diagnostics Collector searches for the files to collect in the directories listed for the **-stamp** option.

Collecting diagnostic data from Java runtime problems

The Diagnostics Collector produces an output file for each problem event that occurs in your Java application.

When you add the command-line option **-Xdiagnosticscollector**, the Diagnostics Collector runs and produces several output .zip files. One file is produced at startup. Another file is produced for each dump event that occurs during the lifetime of the JVM. For each problem event that occurs in your Java application, one .zip file is created to hold all the diagnostic data for that event. For example, an application might have multiple `OutOfMemoryErrors` but keep on running. Diagnostics Collector produces multiple .zip files, each holding the diagnostic data from one `OutOfMemoryError`.

The output .zip file is written to the current working directory by default. You can specify a different location by setting the `output.dir` property in the settings file, as described in "Diagnostics Collector settings" on page 332. An output .zip file name takes the form:

```
java.<event>.<YYYYMMDD.hhmmss.pid>.zip
```

In this file name, *<event>* is one of the following names:

- abortsignal
- check
- dumpevent
- gpf
- outofmemoryerror
- usersignal
- vmstart
- vmstop

These event names refer to the event that triggered Diagnostics Collector. The name provides a hint about the type of problem that occurred. The default name is *dumpevent*, and is used when a more specific name cannot be given for any reason.

<*YYYYMMDD.hhmmss.pid*> is a combination of the time stamp of the dump event, and the process ID for the original Java application. *pid* is not the process ID for the Diagnostics Collector.

The Diagnostics Collector copies files that it writes to the output .zip file. It does not delete the original diagnostic information.

When the Diagnostics Collector finds a system dump for the problem event, then by default it runs **jextract** to post-process the dump and gather context information This information enables later debugging. Diagnostics Collector automates a manual step that is requested by IBM support on most platforms. You can prevent Diagnostics Collector from running **jextract** by setting the property **run.jextract** to **false** in the settings file. For more information, see "Diagnostics Collector settings" on page 332.

The Diagnostics Collector logs its actions and messages in a file named `JavaDiagnosticsCollector.<number>.log`. The log file is written to the current working directory. The log file is also stored in the output .zip file. The <*number*> component in the log file name is not significant; it is added to keep the log file names unique.

The Diagnostics Collector is a Java VM dump agent. It is run by the Java VM in response to the dump events that produce diagnostic files by default. It runs in a new Java process, using the same version of Java as the VM producing dumps. This ensures that the tool runs the correct version of **jextract** for any system dumps produced by the original Java process.

Verifying your Java diagnostics configuration

When you enable the command-line option `-Xdiagnosticscollector`, a diagnostic configuration check runs at Java VM start. If any settings disable the collection of key Java diagnostic data, a warning is reported.

The aim of the diagnostic configuration check is to avoid the situation where a problem occurs after a long time, but diagnostic data is missing because the collection of diagnostic data was inadvertently switched off. Diagnostic configuration check warnings are reported on `stderr` and in the Diagnostics Collector log file. A copy of the log file is stored in the `java.check.<timestamp>.<pid>.zip` output file.

If you do not see any warning messages, it means that the Diagnostics Collector has not found any settings that disable diagnostic data collection. The Diagnostics Collector log file stored in `java.check.<timestamp>.<pid>.zip` gives the full record of settings that have been checked.

For extra thorough checking, the Diagnostics Collector can trigger a Java dump. The dump provides information about the command-line options and current Java system properties. It is worth running this check occasionally, as there are command-line options and Java system properties that can disable significant parts of the Java diagnostic data set. To enable the use of a Java dump for diagnostic configuration checking, set the **config.check.javacore** option to **true** in the settings file. For more information, see "Diagnostics Collector settings" on page 332.

For all platforms, the diagnostic configuration check examines environment variables that can disable Java diagnostic data collection. For reference purposes, the full list of current environment variables and their values is stored in the Diagnostics Collector log file.

Configuring the Diagnostics Collector

The Diagnostics Collector supports various options that can be set in a properties file.

Diagnostics Collector can be configured by using options that are set in a properties file. By default, the properties file is `jre/lib/dc.properties`. If you do not have access to edit this file, or if you are working on a shared system, you can specify an alternative filename using:

```
-Xdiagnosticscollector:settings=<filename>
```

Using a settings file is optional. By default, Diagnostics Collector gathers all the main types of Java diagnostic files.

Diagnostics Collector settings:

The Diagnostics Collector has several settings that affect the way the collector works.

The settings file uses the standard Java properties format. It is a text file with one **property=value** pair on each line. Each supported property controls the Diagnostics Collector in some way. Lines that start with '#' are comments.

Parameters

file.<any_string>=<pathname>

Any property with a name starting **file.** specifies the path to a diagnostic file to collect. You can add any string as a suffix to the property name, as a reminder of which file the property refers to. You can use any number of **file.** properties, so you can tell the Diagnostics Collector to collect a list of custom diagnostic files for your environment. Using **file.** properties does not alter or prevent the collection of all the standard diagnostic files. Collection of standard diagnostic files always takes place.

Custom debugging scripts or software can be used to produce extra output files to help diagnose a problem. In this situation, the settings file is used to identify the extra debug output files for the Diagnostics Collector. The Diagnostics Collector collects the extra debug files at the point when a problem occurs. Using the Diagnostics Collector in this way means that debug files are collected immediately after the problem event, increasing the chance of capturing relevant context information.

output.dir=<output_directory_path>

The Diagnostics Collector tries to write its output .zip file to the output directory path that you specify. The path can be absolute or relative to the working directory of the Java process. If the directory does not exist, the Diagnostics Collector tries to create it. If the directory cannot be created, or the directory is not writeable, the Diagnostics Collector defaults to writing its output .zip file to the current working directory.

loglevel.file=<level>
This setting controls the amount of information written to the Diagnostics Collector log file. The default setting for this property is **config**. Valid levels are:

off No information reported.

severe Errors are reported.

warning
Report warnings in addition to information reported by **severe**.

info More detailed information in addition to that reported by **warning**.

config Configuration information reported in addition to that reported by **info**. This is the default reporting level.

fine Tracing information reported in addition to that reported by **config**.

finer Detailed tracing information reported in addition to that reported by **fine**.

finest Report even more tracing information in addition to that reported by **finer**.

all Report everything.

loglevel.console=<level>
Controls the amount of information written by the Diagnostics Collector to `stderr`. Valid values for this property are as described for loglevel.file. The default setting for this property is **warning**.

settings.id=<identifier>
Allows you to set an identifier for the settings file. If you set loglevel.file to **fine** or **lower**, the **settings.id** is recorded in the Diagnostics Collector log file as a way to check that your settings file is loaded as expected.

config.check.javacore={true|false}
Set **config.check.javacore=true** to enable a Java dump for the diagnostic configuration check at virtual machine start-up. The check means that the virtual machine start-up takes more time, but it enables the most thorough level of diagnostic configuration checking.

run.jextract=false
Set this option to prevent the Diagnostics Collector running **jextract** on detected System dumps.

Known limitations
There are some known limitations for the Diagnostics Collector.

If Java programs do not start at all on your system, for example because of a Java runtime installation problem or similar issue, the Diagnostics Collector cannot run.

The Diagnostics Collector does not respond to additional **-Xdump** settings that specify extra dump events requiring diagnostic information. For example, if you use **-Xdump** to produce dumps in response to a particular exception being thrown, the Diagnostics Collector does not collect the dumps from this event.

Garbage Collector diagnostic data
This section describes how to diagnose garbage collection.

The topics that are discussed in this chapter are:

-
-
- “Verbose garbage collection logging”
- “-Xtgc tracing” on page 337

Verbose garbage collection logging

Verbose logging is intended as the first tool to be used when attempting to diagnose garbage collector problems; you can perform more detailed analysis by calling one or more -Xtgc (trace garbage collector) traces.

Note: The output provided by **-verbose:gc** can and does change between releases. Ensure that you are familiar with details of the different collection strategies by reading “Memory management” on page 23 if necessary.

By default, **-verbose:gc** output is written to stderr. You can redirect the output to a file using the **-Xverbosegclog** command-line option (see “Garbage Collector command-line options” on page 453 for more information). If you redirect the output to a file, you can later analyze the file contents by using IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer. For more information about this tool, see “Using the IBM Monitoring and Diagnostic Tools for Java” on page 219.

In this release, the 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.

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.

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 JVM, verbose logging records the event.

Stop-the-world operations are shown within the `<exclusive-start>` and `<exclusive-end>` tags. The `<exclusive-start>` tag includes the `response-info` section, which provides details about the process of acquiring exclusive access to the JVM.

```
<exclusive-start id="2" timestamp="2010-11-23T00:41:32.515">
  <response-info timems="0.011" idlems="0.011" threads="0"
lastid="00124F00" lastname="main" />
</exclusive-start>
<exclusive-end id="13" timestamp="2010-11-23T00:41:32.517" />
```

Garbage collection cycle:

Verbose garbage collection output shows each garbage collection cycle enclosed within `<cycle-start>` and `<cycle-end>` tags.

The `<cycle-end>` tag contains a `context-id` attribute that indicates 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="2010-11-23T00:41:32.
515" />
```

In the example shown, the <cycle-end> tag has a context-id of 4, which reflects the id value shown for <cycle-start>.

Garbage collection increment:

A complete garbage collection increment is shown within <gc-start> and <gc-end> tags in the verbose output.

The <gc-start> and <gc-end> tags represent the start and end of a garbage collection increment. Both of these tags include a mem-info section. This section includes information about the current state of the Java heap, and any memory used by a specific garbage collection policy.

The <mem type="nursery"> tag shows the amount of free space and total space that is used in the nursery area before and after a scavenge event. The used space is where nursery objects reside and future allocations occur. This space does not account for reserved space in the nursery that is used for flipping survived objects on the next scavenge. The real total nursery size can be calculated as:

```
reported-total-nursery-size/tilt-ratio
```

.

The <mem type="tenure"> tag shows the amount of free space and total space that is used in the tenure area before and after a GC event.

The <mem-info> tag shows the cumulative amount of free and total used space in the heap. Similarly, as with total nursery size, this value does not account for survivor space in the nursery. The real total heap size can be calculated as:

```
reported-total-tenure-heap-size + reported-total-nursery-size/tilt-ratio
```

The <gc-start> and <gc-end> tags contain a context-id attribute that indicates the id of the corresponding garbage collection cycle.

```
<gc-start id="5" type="scavenge" contextid="4" timestamp="2010-11-23T00:41:32.515">
  <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-end id="8" type="scavenge" contextid="4" durationms="2.204" timestamp="2010-11-
 23T00:41:32.517">
  <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>
```

In the example, the contextid of 4 tells you that this garbage collection increment is part of the garbage collection cycle that has the tag <cycle-start id="4">.

Garbage collection operation:

Every garbage collection increment contains at least one garbage collection operation, shown in the verbose output with a <gc-op> tag.

The timems attribute of the <gc-op> tag indicates the time taken to complete the garbage collection operation, in milliseconds. The contextid attribute indicates the ID of the corresponding garbage collection cycle. In the following example, contextid="4" indicates that this garbage collection operation is part of the garbage collection cycle that has the tag <cycle-start id="4">.

The <gc-op> output contains subsections that describe operations that are specific to different garbage collection policies. These subsections might change from release to release, when improvements are made to the technology or when new data becomes available.

```
<gc-op id="7" type="scavenge" timems="1.127" contextid="4" timestamp="2010-11-23T00
 :41:32.515">
  <scavenger-info tenureage="10" tiltratio="50" />
  <memory-copied type="nursery" objects="2304" bytes="289896" bytesdiscarded="0" />
  <finalization candidates="28" enqueued="1" />
  <references type="soft" candidates="5" cleared="0" enqueued="0" dynamicThreshold=
   "32" maxThreshold="32" />
  <references type="weak" candidates="6" cleared="4" enqueued="0" />
  <references type="phantom" candidates="1" cleared="0" enqueued="0" />
</gc-op>
```

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.

```
<af-start id="3" totalBytesRequested="12936" timestamp="2010-11-23T00:41:32.515"
intervalms="225.036" />

<allocation-satisfied id="11" thread="00124F00" bytesRequested="12936" />
<af-end id="12" timestamp="2010-11-23T00:41:32.515"/>
```

-Xtgc tracing

By enabling one or more TGC (trace garbage collector) traces, more detailed garbage collection information than that displayed by **-verbose:gc** will be shown.

This section summarizes the different **-Xtgc** traces available. The output is written to stdout. More than one trace can be enabled simultaneously by separating the parameters with commas, for example **-Xtgc:backtrace,compaction**.

-Xtgc:backtrace:

This trace shows information tracking which thread triggered the garbage collection.

For a System.gc() this might be similar to:

```
"main" (0x0003691C)
```

This shows that the GC was triggered by the thread with the name "main" and osThread `0x0003691C`.

One line is printed for each global or scavenger collection, showing the thread that triggered the GC.

-Xtgc:compaction:

This trace shows information relating to compaction.

The trace is similar to:

```
Compact(3): reason = 7 (forced compaction)
Compact(3): Thread 0, setup stage: 8 ms.
Compact(3): Thread 0, move stage: handled 42842 objects in 13 ms, bytes moved 2258028.
Compact(3): Thread 0, fixup stage: handled 0 objects in 0 ms, root fixup time 1 ms.
Compact(3): Thread 1, setup stage: 0 ms.
Compact(3): Thread 1, move stage: handled 35011 objects in 8 ms, bytes moved 2178352.
Compact(3): Thread 1, fixup stage: handled 74246 objects in 13 ms, root fixup time 0 ms.
Compact(3): Thread 2, setup stage: 0 ms.
Compact(3): Thread 2, move stage: handled 44795 objects in 32 ms, bytes moved 2324172.
Compact(3): Thread 2, fixup stage: handled 6099 objects in 1 ms, root fixup time 0 ms.
Compact(3): Thread 3, setup stage: 8 ms.
Compact(3): Thread 3, move stage: handled 0 objects in 0 ms, bytes moved 0.
Compact(3): Thread 3, fixup stage: handled 44797 objects in 7 ms, root fixup time 0 ms.
```

This trace shows that compaction occurred during the third global GC, for reason "7". In this case, four threads are performing compaction. The trace shows the work performed by each thread during setup, move, and fixup. The time for each stage is shown together with the number of objects handled by each thread.

-Xtgc:concurrent:

This trace displays basic extra information about the concurrent mark helper thread.

Note: You cannot use this option with the **-Xgcpolicy:balanced** option. If you attempt to use these two options together, the JVM does not start.

```
<CONCURRENT GC BK thread 0x0002645F activated after GC(5)>

<CONCURRENT GC BK thread 0x0002645F (started after GC(5)) traced 25435>
```

This trace shows when the background thread was activated, and the amount of tracing it performed (in bytes).

-Xtgc:dump:

This trace shows extra information following the sweep phase of a global garbage collection.

This is an extremely large trace – a sample of one GC's output is:

```
<GC(4) 13F9FE44 freelen=x000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA0140 freelen=x00000010>
<GC(4) 13FA0150 freelen=x00000050 -- x0000001C java/lang/String>
<GC(4) 13FA0410 freelen=x000002C4 -- x00000024 spec/jbb/infra/Collections/
     longBTreeNode>
<GC(4) 13FA0788 freelen=x00000004 -- x00000050 java/lang/Object[]>
```

```
<GC(4) 13FA0864 freelen=x00000010>
<GC(4) 13FA0874 freelen=x0000005C -- x0000001C java/lang/String>
<GC(4) 13FA0B4C freelen=x000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA0E48 freelen=x00000010>
<GC(4) 13FA0E58 freelen=x00000068 -- x0000001C java/lang/String>
<GC(4) 13FA1148 freelen=x000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA1444 freelen=x00000010>
<GC(4) 13FA1454 freelen=x0000006C -- x0000001C java/lang/String>
<GC(4) 13FA174C freelen=x000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA1A48 freelen=x00000010>
<GC(4) 13FA1A58 freelen=x00000054 -- x0000001C java/lang/String>
<GC(4) 13FA1D20 freelen=x000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA201C freelen=x00000010>
<GC(4) 13FA202C freelen=x00000044 -- x0000001C java/lang/String>
<GC(4) 13FA22D4 freelen=x000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA25D0 freelen=x00000010>
<GC(4) 13FA25E0 freelen=x00000048 -- x0000001C java/lang/String>
<GC(4) 13FA2890 freelen=x000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA2B8C freelen=x00000010>
<GC(4) 13FA2B9C freelen=x00000068 -- x0000001C java/lang/String>
<GC(4) 13FA2E8C freelen=x000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA3188 freelen=x00000010>
```

A line of output is printed for every free chunk in the system, including dark matter (free chunks that are not on the free list for some reason, usually because they are too small). Each line contains the base address and the size in bytes of the chunk. If the chunk is followed in the heap by an object, the size and class name of the object is also printed.

-Xtgc:excessiveGC:

This trace shows statistics for garbage collection cycles.

After a garbage collection cycle has completed, a trace entry is produced:

```
excessiveGC: gcid="10" intimems="122.269" outtimems="1.721" \
             percent="98.61" averagepercent="37.89"
```

This trace shows how much time was spent performing garbage collection and how much time was spent out of garbage collection. In this example, garbage collection cycle 10 took 122.269 ms to complete and 1.721 ms passed between collections 9 and 10. These statistics show that garbage collection accounted for 98.61% of the time from the end of collection 9 to the end of collection 10. The average time spent in garbage collection is 37.89%.

When the average time in garbage collection reaches 95%, extra trace entries are produced:

```
excessiveGC: gcid="65" percentreclaimed="1.70" freedelta="285728" \
             activesize="16777216" currentsize="16777216" maxiumumsize="16777216"
```

This trace shows how much garbage was collected. In this example, 285728 bytes were reclaimed by garbage collection 65, which accounts for 1.7% of the total heap size. The example also shows that the heap has expanded to its maximum size (see **-Xmx** in “Garbage Collector command-line options” on page 453).

When the average time in garbage collection reaches 95% and the percentage of free space reclaimed by a collection drops below 3%, another trace entry is produced:

```
excessiveGC: gcid="65" percentreclaimed="1.70" minimum="3.00" excessive gc raised
```

The JVM will then throw an OutOfMemoryError.

-Xtgc:freelist:

Before a garbage collection, this trace prints information about the free list and allocation statistics since the last GC.

The trace prints the number of items on the free list, including "deferred" entries (with the scavenger, the unused semispace is a deferred free list entry). For TLH and non-TLH allocations, this prints the total number of allocations, the average allocation size, and the total number of bytes discarded during allocation. For non-TLH allocations, also included is the average number of entries that were searched before a sufficiently large entry was found.

```
 *8* free         0
  *8* deferred     0
total               0
<Alloc TLH: count 3588, size 3107, discard 31>
<  non-TLH: count 6219, search 0, size 183, discard 0>
```

-Xtgc:parallel:

This trace shows statistics about the activity of the parallel threads during the mark and sweep phases of a global garbage collection.

```
Mark:   busy  stall  tail   acquire   release
   0:      30    30     0         0         3
   1:      53     7     0        91        94
   2:      29    31     0        37        37
   3:      37    24     0       243       237
Sweep:  busy   idle sections 127  merge 0
   0:      10     0    96
   1:       8     1     0
   2:       8     1    31
   3:       8     1     0
```

This trace shows four threads (0-3), together with the work done by each thread during the mark and sweep phases of garbage collection.

For the mark phase of garbage collection, the time spent in the "busy", "stalled", and "tail" states is shown (in milliseconds). The number of work packets each thread acquired and released during the mark phase is also shown.

For the sweep phase of garbage collection, the time spent in the "busy" and "idle" states is shown (in milliseconds). The number of sweep chunks processed by each thread is also shown, including the total (127). The total merge time is also shown (0ms).

-Xtgc:scavenger:

This trace prints a histogram following each scavenger collection.

Note: You cannot use this option with the **-Xgcpolicy:balanced** option. If you attempt to use these two options together, the JVM does not start.

A graph is shown of the different classes of objects remaining in the survivor space, together with the number of occurrences of each class and the age of each object (the number of times it has been flipped). A sample of the output from a single scavenge is shown as follows:

```
{SCAV: tgcScavenger OBJECT HISTOGRAM}

{SCAV: | class | instances of age 0-14 in semi-space |
{SCAV: java/lang/ref/SoftReference 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/io/FileOutputStream 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: sun/nio/cs/StreamEncoder$ConverterSE 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/io/FileInputStream 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: char[][] 0 102 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/lang/ref/SoftReference[] 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/io/BufferedOutputStream 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/io/BufferedWriter 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/io/OutputStreamWriter 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/io/PrintStream 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/io/BufferedInputStream 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/lang/Thread[] 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/lang/ThreadGroup[] 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: sun/io/ByteToCharCp1252 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: sun/io/CharToByteCp1252 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
```

-Xtgc:terse:

This trace dumps the contents of the entire heap before and after a garbage collection.

This is an extremely large trace. For each object or free chunk in the heap, a line of trace output is produced. Each line contains the base address, "a" if it is an allocated object and "f" if it is a free chunk, the size of the chunk in bytes, and if it is an object, its class name. A sample is shown as follows:

```
*DH(1)* 230AD778 a x0000001C java/lang/String
*DH(1)* 230AD794 a x00000048 char[]
*DH(1)* 230AD7DC a x00000018 java/lang/StringBuffer
*DH(1)* 230AD7F4 a x00000030 char[]
*DH(1)* 230AD824 a x00000054 char[]
*DH(1)* 230AD878 a x0000001C java/lang/String
*DH(1)* 230AD894 a x00000018 java/util/HashMapEntry
*DH(1)* 230AD8AC a x0000004C char[]
*DH(1)* 230AD8F8 a x0000001C java/lang/String
*DH(1)* 230AD914 a x0000004C char[]
*DH(1)* 230AD960 a x00000018 char[]
*DH(1)* 230AD978 a x0000001C java/lang/String
*DH(1)* 230AD994 a x00000018 char[]
*DH(1)* 230AD9AC a x00000018 java/lang/StringBuffer
*DH(1)* 230AD9C4 a x00000030 char[]
*DH(1)* 230AD9F4 a x00000054 char[]
*DH(1)* 230ADA48 a x0000001C java/lang/String
*DH(1)* 230ADA64 a x00000018 java/util/HashMapEntry
*DH(1)* 230ADA7C a x00000050 char[]
*DH(1)* 230ADACC a x0000001C java/lang/String
*DH(1)* 230ADAE8 a x00000050 char[]
*DH(1)* 230ADB38 a x00000018 char[]
*DH(1)* 230ADB50 a x0000001C java/lang/String
*DH(1)* 230ADB6C a x00000018 char[]
*DH(1)* 230ADB84 a x00000018 java/lang/StringBuffer
*DH(1)* 230ADB9C a x00000030 char[]
*DH(1)* 230ADBCC a x00000054 char[]
*DH(1)* 230ADC20 a x0000001C java/lang/String
*DH(1)* 230ADC3C a x00000018 java/util/HashMapEntry
*DH(1)* 230ADC54 a x0000004C char[]
```

Class-loader diagnostic data

There is some diagnostic data that is available for class-loading.

The topics that are discussed in this chapter are:

- "Class-loader command-line options"
- "Class-loader runtime diagnostic data"
- "Loading from native code" on page 343

Class-loader command-line options

There are some extended command-line options that are available

These options are:

-verbose:dynload
Provides detailed information as each class is loaded by the JVM, including:

- The class name and package.
- For class files that were in a .jar file, the name and directory path of the .jar (for bootstrap classes only).
- Details of the size of the class and the time taken to load the class.

The data is written out to stderr. An example of the output follows:

```
<Loaded java/lang/String from C:\sdk\jre\lib\vm.jar>
<Class size 17258; ROM size 21080; debug size 0>
<Read time 27368 usec; Load time 782 usec; Translate time 927 usec>
```

-Xfuture
Turns on strict class-file format checks. Use this flag when you are developing new code because stricter checks will become the default in future releases. By default, strict format checks are disabled.

-Xverify[:<option>]
With no parameters, enables the Java bytecode verifier, which is the default. Therefore, if used on its own with no parameters, the option has no effect. Optional parameters are:

- **all** - enable maximum verification
- **none** - disable the verifier
- **remote** - enables strict class-loading checks on remotely loaded classes

The verifier is on by default and must be enabled for all production servers. Running with the verifier off, is not a supported configuration. If you encounter problems and the verifier was turned off using **-Xverify:none**, remove this option and try to reproduce the problem.

Class-loader runtime diagnostic data

Use the command-line parameter **-Dibm.cl.verbose=**<*class_expression*> to enable you to trace the way the class loaders find and load application classes.

Alternatively, you can use IBM Monitoring and Diagnostic Tools for Java - Health Center to monitor the application and view class loading information such as when the class was loaded, whether the class was loaded from the shared cache, the number of instances of the class, and the amount of heap space that those instances are occupying.

Here is an example of the **-Dibm.cl.verbose** parameter:

```
C:\j9test>java -Dibm.cl.verbose=*HelloWorld hw.HelloWorld
```

This example produces output that is similar to this:

```
ExtClassLoader attempting to find hw.HelloWorld
ExtClassLoader using classpath C:\sdk\jre\lib\ext\CmpCrmf.jar;C:\sdk\jre\lib\ext\dtfj-interface.jar;
C:\sdk\jre\lib\ext\dtfj.jar;C:\sdk\jre\lib\ext\gskikm.jar;C:\sdk\jre\lib\ext\ibmcmsprovider.jar;C:\s
```

```
dk\jre\lib\ext\ibmjcefips.jar;C:\sdk\jre\lib\ext\ibmjceprovider.jar;C:\sdk\jre\lib\ext\ibmkeycert.ja
r;C:\sdk\jre\lib\ext\IBMKeyManagementServer.jar;C:\sdk\jre\lib\ext\ibmpkcs11.jar;C:\sdk\jre\lib\ext\
ibmpkcs11impl.jar;C:\sdk\jre\lib\ext\ibmsaslprovider.jar;C:\sdk\jre\lib\ext\indicim.jar;C:\sdk\jre\l
ib\ext\jaccess.jar;C:\sdk\jre\lib\ext\JawBridge.jar;C:\sdk\jre\lib\ext\jdmpview.jar
ExtClassLoader path element C:\sdk\jre\lib\ext\CmpCrmf.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\dtfj-interface.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\dtfj.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\gskikm.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\ibmcmsprovider.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\ibmjcefips.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\ibmjceprovider.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\ibmkeycert.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\IBMKeyManagementServer.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\ibmpkcs11.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\ibmpkcs11impl.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\ibmsaslprovider.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\indicim.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\jaccess.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\JawBridge.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\jdmpview.jar does not exist
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\CmpCrmf.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\dtfj-interface.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\dtfj.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\gskikm.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\ibmcmsprovider.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\ibmjcefips.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\ibmjceprovider.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\ibmkeycert.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\IBMKeyManagementServer.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\ibmpkcs11.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\ibmpkcs11impl.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\ibmsaslprovider.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\indicim.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\jaccess.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\JawBridge.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\jdmpview.jar
ExtClassLoader could not find hw.HelloWorld

AppClassLoader attempting to find hw.HelloWorld
AppClassLoader using classpath C:\j9test
AppClassLoader path element C:\j9test does not exist
AppClassLoader found hw/HelloWorld.class in C:\j9test
AppClassLoader found hw.HelloWorld
```

The sequence of the loaders' output is a result of the “delegate first” convention of class loaders. In this convention, each loader checks its cache and then delegates to its parent loader. Then, if the parent returns null, the loader checks the file system or equivalent. This part of the process is reported in the previous example.

The *<class_expression>* can be given as any Java regular expression. “Dic*” matches all classes with names begins with “Dic”, and so on.

Loading from native code

A class loader loads native libraries for a class.

Class loaders look for native libraries in different places:

- If the class that makes the native call is loaded by the Bootstrap class loader, this loader looks in the path that is specified by the `sun.boot.library.path` property, to load the libraries.
- If the class that makes the native call is loaded by the Extensions class loader, this loader looks in the paths that are specified by the following properties, in this order:

1. `java.ext.dirs`
2. `sun.boot.library.path`
3. `java.library.path`

- If the class that makes the native call is loaded by the Application class loader, this loader looks in the paths that are specified by the following properties, in this order:
  1. `sun.boot.library.path`
  2. `java.library.path`
- If the class that makes the native call is loaded by a Custom class loader, this loader defines the search path to load libraries.

Shared classes diagnostic data

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

For an introduction to shared classes, see "Class data sharing" on page 55.

The topics that are discussed in this chapter are:

Deploying shared classes

You cannot enable class sharing without considering how to deploy it sensibly for your application. This section looks at some of the important issues to consider.

Cache naming:

If multiple users will be using an application that is sharing classes or multiple applications are sharing the same cache, knowing how to name caches appropriately is important. The ultimate goal is to have the smallest number of caches possible, while maintaining secure access to the class data and allowing as many applications and users as possible to share the same classes.

To use a cache for a specific application, write the cache into the application installation directory, or a directory within that directory, using the **-Xshareclasses:cachedir=**<*dir*> suboption. This helps prevent users of other applications from accidentally using the same cache, and automatically removes the cache if the application is uninstalled. If the directory does not exist it is created.

If you specify a directory that does not already exist, you can use the **-Xshareclasses:cacheDirPerm=**<*permission*> suboption to specify permissions for the directory when it is created. 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. For more information about the `cacheDirPerm` suboption, see "JVM command-line options" on page 428.

If the same user will always be using the same application, either use the default cache name (which includes the user name) or specify a cache name specific to the application. The user name can be incorporated into a cache name using the %u modifier, which causes each user running the application to get a separate cache.

If multiple users in the same operating system group are running the same application, use the **groupAccess** suboption, which creates the cache allowing all users in the same primary group to share the same cache. If multiple operating system groups are running the same application, the %g modifier can be added to the cache name, causing each group running the application to get a separate cache.

Multiple applications or different JVM installations can share the same cache provided that the JVM installations are of the same service release level. It is possible for different JVM service releases to share the same cache, but it is not advised. The JVM will attempt to destroy and re-create a cache created by a different service release. See "Compatibility between service releases" on page 351 for more information.

Small applications that load small numbers of application classes should all try to share the same cache, because they will still be able to share bootstrap classes. For large applications that contain completely different classes, it might be more sensible for them to have a class cache each, because there will be few common classes and it is then easier to selectively clean up caches that aren't being used.

The default directory is `/tmp`, which is shared by all users.

Cache access:

A JVM can access a shared class cache with either read-write or read-only access. Read-write access is the default and gives all users equal rights to update the cache. Use the **`-Xshareclasses:readonly`** option for read-only access.

Opening a cache as read-only makes it easier to administer operating system permissions. A cache created by one user cannot be opened read-write by other users, but other users can reduce startup time by opening the cache as read-only. Opening a cache as read-only also prevents corruption of the cache. This option can be useful on production systems where one instance of an application corrupting the cache might affect the performance of all other instances.

When a cache is opened read-only, class files of the application that are modified or moved cannot be updated in the cache. Sharing is disabled for the modified or moved containers for that JVM.

Cache housekeeping:

Unused caches on a system waste resources that might be used by another application. Ensuring that caches are sensibly managed is important.

The JVM offers a number of features to assist in cache housekeeping. To understand these features, it is important to explain the differences in behavior between persistent and non-persistent caches.

Persistent caches are written to disk and remain there until explicitly removed. Persistent caches are not removed when the operating system is restarted. Because

persistent caches do not exist in shared memory, the only penalty of not removing stale caches is that they take up disk space.

Non-persistent caches exist in shared memory and retain system resources that might be used by other applications. However, non-persistent caches are automatically purged when the operating system is restarted, so housekeeping is only an issue between operating system restarts.

To perform housekeeping functions successfully, whether automatically or explicitly, you must have the correct operating system permissions. In general, if a user has the permissions to open a cache with read-write access, they also have the permissions to remove it. The only exception is for non-persistent caches. These caches can be removed only by the user that created the cache. Caches can only be removed if they are not in use.

The JVM provides a number of housekeeping utilities, which are all suboptions to the **-Xshareclasses** command-line option. Each suboption performs the explicit action requested. The suboption might also perform other automated housekeeping activities. Each suboption works in the context of a specific **cacheDir**.

destroy
: This suboption removes all the generations of a named cache. The term "generation" means all caches created by earlier or later service releases or versions of the JVM.

destroyAll
: This suboption removes all caches in the specified **cacheDir**.

expire=<*time in minutes*>
: This suboption looks for caches which have not been connected to for the <*time in minutes*> specified. If any caches are found which have not been connected to in that specified time, they are removed.

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

expire=10000
: This suboption removes all caches which have not been used for approximately one week.

There is also a certain amount of automatic housekeeping which is done by the JVM. Most of this automatic housekeeping is driven by the cache utilities. **destroyAll** and **expire** attempt to remove all persistent and non-persistent caches of all JVM levels and service releases in a given **cacheDir**. **destroy** only works on a specific cache of a specific name and type.

Cases where the JVM attempts automatic housekeeping when not requested by the user include:

- When a JVM connects to a cache, and determines that the cache is corrupt or was created by a different service release. The JVM attempts to remove and re-create the cache.
- If `/tmp/javasharedresources` is deleted. The JVM attempts to identify any leaked shared memory areas that originate from non-persistent caches. If any areas are found, they are purged.

With persistent caches, it is safe to delete the cache files manually from the file system. Each persistent cache has only one system object: the cache file.

It is not safe to delete cache files manually for non-persistent caches. The reason is that each non-persistent cache has four system objects: A shared memory area, a shared semaphore, and two control files to identify the memory and semaphores to the JVM. Deleting the control files causes the memory and semaphores to be leaked. They can then only be identified and removed using the **ipcs** and **ipcrm** commands.

The **reset** suboption can also be used to cause a JVM to refresh an existing class cache when it starts. All generations of the named cache are removed and the current generation is re-created if it is not already in use. The option **-Xshareclasses:reset** can be added anywhere to the command line. The option does not override any other **Xshareclasses** command-line options. This constraint means that **-Xshareclasses:reset** can be added to the **IBM_JAVA_OPTIONS** environment variable, or any of the other means of passing command-line options to the JVM.

Cache performance:

Shared classes use optimizations to maintain performance under most circumstances. However, there are configurable factors that can affect shared classes performance.

Use of Java archive and compressed files

The cache keeps itself up-to-date with file system updates by constantly checking file system timestamps against the values in the cache.

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

`.class` files can be created or deleted from a directory at any time. If you include a directory name in a classpath, shared classes performance can be affected because the directory is constantly checked for classes. The impact on performance might be greater if the directory name is near the beginning of the classpath string. For example, consider a classpath of `/dir1:jar1.jar:jar2.jar:jar3.jar;`. When loading any class from the cache using this classpath, the directory `/dir1` must be checked for the existence of the class for every class load. This checking also requires fabricating the expected directory from the package name of the class. This operation can be expensive.

Advantages of not filling the cache

A full shared classes cache is not a problem for any JVMs connected to it. However, a full cache can place restrictions on how much sharing can be performed by other JVMs or applications.

ROMClasses are added to the cache and are all unique. Metadata is added describing the ROMClasses and there can be multiple metadata entries corresponding to a single ROMClass. For example, if class A is loaded from myApp1.jar and another JVM loads the same class A from myOtherApp2.jar, only one ROMClass exists in the cache. However there are two pieces of metadata that describe the source locations.

If many classes are loaded by an application and the cache is 90% full, another installation of the same application can use the same cache. The extra information that must be added about the classes from the second application is minimal.

After the extra metadata has been added, both installations can share the same classes from the same cache. However, if the first installation fills the cache completely, there is no room for the extra metadata. The second installation cannot share classes because it cannot update the cache. The same limitation applies for classes that become stale and are redeemed. See "Redeeming stale classes" on page 357. Redeeming the stale class requires a small quantity of metadata to be added to the cache. If you cannot add to the cache, because it is full, the class cannot be redeemed.

Read-only cache access

If the JVM opens a cache with read-only access, it does not obtain any operating system locks to read the data. This behavior can make cache access slightly faster. However, if any containers of cached classes are changed or moved on a classpath, then sharing is disabled for all classes on that classpath. There are two reasons why sharing is disabled:

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

Page protection

By default, the JVM protects all cache memory pages using page protection to prevent accidental corruption by other native code running in the process. If any native code attempts to write to the protected page, the process ends, but all other JVMs are unaffected.

The only page not protected by default is the cache header page, because the cache header must be updated much more frequently than the other pages. The cache header can be protected by using the **-Xshareclasses:mprotect=all** option. This option has a small affect on performance and is not enabled by default.

Switching off memory protection completely using **-Xshareclasses:mprotect=none** does not provide significant performance gains.

Caching Ahead Of Time (AOT) code

The JVM might automatically store a small amount of Ahead Of Time (AOT) compiled native code in the cache when it is populated with classes. The AOT code enables any subsequent JVMs attaching to the cache to start faster. AOT data is generated for methods that are likely to be most effective.

You can use the **-Xshareclasses:noaot**, **-Xscminaot**, and **-Xscmaxaot** options to control the use of AOT code in the cache. See "JVM command-line options" on page 428 for more information.

In general, the default settings provide significant startup performance benefits and use only a small amount of cache space. In some cases, for example, running the JVM without the JIT, there is no benefit gained from the cached AOT code. In these cases, turn off caching of AOT code.

To diagnose AOT issues, use the **-Xshareclasses:verboseAOT** command-line option. This option generates messages when AOT code is found or stored in the cache.

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.

In general, the default settings provide significant performance benefits and use only a small amount of cache space.

Making the most efficient use of cache space

A shared class cache is a finite size and cannot grow. The JVM makes more efficient use of cache space by sharing strings between classes, and ensuring that classes are not duplicated. However, there are also command-line options that optimize the cache space available.

-Xscminaot and **-Xscmaxaot** place maximum and minimum limits on the amount of AOT data the JVM can store in the cache. **-Xshareclasses:noaot** prevents the JVM from storing any AOT data.

-Xscminjitdata<size> and **-Xscmaxjitdata<size>** place maximum and minimum limits on the amount of JIT data the JVM can store in the cache. **-Xshareclasses:nojitdata** prevents the JVM from storing any JIT data.

-Xshareclasses:nobootclasspath disables the sharing of classes on the boot classpath, so that only classes from application class loaders are shared. There are also optional filters that can be applied to Java classloaders to place custom limits on the classes that are added to the cache.

Very long classpaths

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

Matching very long classpaths is initially expensive, but successful and failed matches are remembered. Therefore, loading classes from the cache using very long classpaths is much faster than loading from disk.

Growing classpaths

Where possible, avoid gradually growing a classpath in a URLClassLoader using addURL(). Each time an entry is added, an entire new classpath must be added to the cache.

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

Concurrent access

A shared class cache can be updated and read concurrently by any number of JVMs. Any number of JVMs can read from the cache while a single JVM is writing to it.

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

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

Class GC with shared classes

Running with shared classes has no affect on class garbage collection. Class loaders loading classes from the shared class cache can be garbage collected in the same way as class loaders that load classes from disk. If a class loader is garbage collected, the ROMClasses it has added to the cache persist.

Class Debug Area

A portion of the shared classes cache is reserved for storing the class attribute information LineNumberTable and LocalVariableTable during 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.

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.

Compatibility between service releases:

Use the most recent service release of a JVM for any application.

It is not recommended for different service releases to share the same class cache concurrently. A class cache is compatible with earlier and later service releases. However, there might be small changes in the class files or the internal class file format between service releases. These changes might result in duplication of classes in the cache. For example, a cache created by a given service release can continue to be used by an updated service release, but the updated service release might add extra classes to the cache if space allows.

To reduce class duplication, if the JVM connects to a cache which was created by a different service release, it attempts to destroy the cache then re-create it. This automated housekeeping feature is designed so that when a new JVM level is used with an existing application, the cache is automatically refreshed. However, the refresh only succeeds if the cache is not in use by any other JVM. If the cache is in use, the JVM cannot refresh the cache, but uses it where possible.

If different service releases do use the same cache, the JVM disables AOT. The effect is that AOT code in the cache is ignored.

Dealing with runtime bytecode modification

Modifying bytecode at run time is an increasingly popular way to engineer required function into classes. Sharing modified bytecode improves startup time, especially when the modification being used is expensive. You can safely cache modified bytecode and share it between JVMs, but there are many potential problems because of the added complexity. It is important to understand the features described in this section to avoid any potential problems.

This section contains a brief summary of the tools that can help you to share modified bytecode.

Potential problems with runtime bytecode modification:

The sharing of modified bytecode can cause potential problems.

When a class is stored in the cache, the location from which it was loaded and a time stamp indicating version information are also stored. When retrieving a class from the cache, the location from which it was loaded and the time stamp of that location are used to determine whether the class should be returned. The cache does not note whether the bytes being stored were modified before they were defined unless it is specifically told so. Do not underestimate the potential problems that this modification could introduce:

- In theory, unless all JVMs sharing the same classes are using exactly the same bytecode modification, JVMs could load incorrect bytecode from the cache. For example, if JVM1 populates a cache with modified classes and JVM2 is not using a bytecode modification agent, but is sharing classes with the same cache, it could incorrectly load the modified classes. Likewise, if two JVMs start at the same time using different modification agents, a mix of classes could be stored and both JVMs will either throw an error or demonstrate undefined behavior.
- An important prerequisite for caching modified classes is that the modifications performed must be deterministic and final. In other words, an agent which performs a particular modification under one set of circumstances and a different modification under another set of circumstances, cannot use class caching. This is because only one version of the modified class can be cached for any given agent and once it is cached, it cannot be modified further or returned to its unmodified state.

In practice, modified bytecode can be shared safely if the following criteria are met:

- Modifications made are deterministic and final (described previously).
- The cache knows that the classes being stored are modified in a particular way and can partition them accordingly.

The VM provides features that allow you to share modified bytecode safely, for example using "modification contexts". However, if a JVMTI agent is unintentionally being used with shared classes without a modification context, this usage does not cause unexpected problems. In this situation, if the VM detects the presence of a JVMTI agent that has registered to modify class bytes, it forces all bytecode to be loaded from disk and this bytecode is then modified by the agent. The potentially modified bytecode is passed to the cache and the bytes are compared with known classes of the same name. If a matching class is found, it is reused; otherwise, the potentially modified class is stored in such a way that other JVMs cannot load it accidentally. This method of storing provides a "safety net" that ensures that the correct bytecode is always loaded by the JVM running the agent, but any other JVMs sharing the cache will be unaffected. Performance during class loading could be affected because of the amount of checking involved, and because bytecode must always be loaded from disk. Therefore, if modified bytecode is being intentionally shared, the use of modification contexts is recommended.

Modification contexts:

A modification context creates a private area in the cache for a given context, so that multiple copies or versions of the same class from the same location can be stored using different modification contexts. You choose the name for a context, but it must be consistent with other JVMs using the same modifications.

For example, one JVM uses a JVMTI agent "agent1", a second JVM uses no bytecode modification, a third JVM also uses "agent1", and a fourth JVM uses a different agent, "agent2". If the JVMs are started using the following command lines (assuming that the modifications are predictable as described previously), they should all be able to share the same cache:

```
java -agentlib:agent1 -Xshareclasses:name=cache1,modified=myAgent1 myApp.ClassName
java -Xshareclasses:name=cache1 myApp.ClassName
java -agentlib:agent1 -Xshareclasses:name=cache1,modified=myAgent1 myApp.ClassName
java -agentlib:agent2 -Xshareclasses:name=cache1,modified=myAgent2 myApp.ClassName
```

SharedClassHelper partitions:

Modification contexts cause all classes loaded by a particular JVM to be stored in a separate cache area. If you need a more granular approach, the SharedClassHelper API can store individual classes under "partitions".

This ability to use partitions allows an application class loader to have complete control over the versioning of different classes and is particularly useful for storing bytecode woven by Aspects. A partition is a string key used to identify a set of classes. For example, a system might weave a number of classes using a particular Aspect path and another system might weave those classes using a different Aspect path. If a unique partition name is computed for the different Aspect paths, the classes can be stored and retrieved under those partition names.

The default application class loader or bootstrap class loader does not support the use of partitions; instead, a SharedClassHelper must be used with a custom class loader.

Using the 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, as described in "Potential problems with runtime bytecode modification" on page 352. 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 347.

JVMTI redefinition and retransformation of classes:

Redefined classes are never stored in the cache. Retransformed classes are not stored in the cache by default, but caching can be enabled using the **-Xshareclasses:cacheRetransformed** option.

Redefined classes are classes containing replacement bytecode provided by a JVMTI agent at run time, typically where classes are modified during a debugging session. Redefined classes are never stored in the cache.

Retransformed classes are classes with registered retransformation capable agents that have been called by a JVMTI agent at run time. Unlike RedefineClasses, the RetransformClasses function allows the class definition to be changed without reference to the original bytecode. An example of retransformation is a profiling agent that adds or removes profiling calls with each retransformation. Retransformed classes are not stored in the cache by default, but caching can be enabled using the **-Xshareclasses:cacheRetransformed** option. This option will also work with modification contexts or partitions.

Further considerations for runtime bytecode modification:

There are a number of additional items that you need to be aware of when using the cache with runtime bytecode modification.

If bytecode is modified by a non-JVMTI agent and defined using the JVM's application class loader when shared classes are enabled, these modified classes are stored in the cache and nothing is stored to indicate that these are modified classes. Another JVM using the same cache will therefore load the classes with these modifications. If you are aware that your JVM is storing modified classes in the cache using a non-JVMTI agent, you are advised to use a modification context with that JVM to protect other JVMs from the modifications.

Combining partitions and modification contexts is possible but not recommended, because you will have partitions inside partitions. In other words, a partition A stored under modification context X will be different from partition A stored under modification context B.

Because the shared class cache is a fixed size, storing many different versions of the same class might require a much larger cache than the size that is typically required. However, note that the identical classes are never duplicated in the cache, even across modification contexts or partitions. Any number of metadata entries might describe the class and where it came from, but they all point to the same class bytes.

If an update is made to the file system and the cache marks a number of classes as stale as a result, note that it will mark all versions of each class as stale (when versions are stored under different modification contexts or partitions) regardless of the modification context being used by the JVM that caused the classes to be marked stale.

Understanding dynamic updates

The shared class cache must respond to file system updates; otherwise, a JVM might load classes from the cache that are out of date or "stale". After a class has been marked stale, it is not returned by the cache if it is requested by a class loader. Instead, the class loader must reload the class from disk and store the updated version in the cache.

The cache is managed in a way that helps ensure that the following challenges are addressed:

- Java archive and compressed files are usually locked by class loaders when they are in use. The files can be updated when the JVM shuts down. Because the cache persists beyond the lifetime of any JVM using it, subsequent JVMs connecting to the cache check for Java archive and compressed file updates.
- `.class` files that are not in a `.jar` file can be updated at any time during the lifetime of a JVM. The cache checks for individual class file updates.
- `.class` files can be created or removed from directories found in classpaths at any time during the lifetime of a JVM. The cache checks the classpath for classes that have been created or removed.
- `.class` files must be in a directory structure that reflects their package structure. This structure helps ensure that when checking for updates, the correct directories are searched.

Class files contained in .jar files and compressed files, and class files stored as `.class` files on the file system, are accessed and used in different ways. The result

is that the cache treats them as two different types. Updates are managed by writing file system time stamps into the cache.

Classes found or stored using a SharedClassTokenHelper cannot be maintained in this way, because Tokens are meaningless to the cache. As a direct consequence of updating the class data, AOT data is automatically updated.

Storing classes

When a classpath is stored in the cache, the Java archive and compressed files are time stamped. These time stamps are stored as part of the classpath. Directories are not time stamped. When a ROMClass is stored, if it came from a `.class` file on the file system, the `.class` file it came from is time stamped and this time stamp is stored. Directories are not time stamped because there is no guarantee that subsequent updates to a file cause an update to the directory holding the file.

If a compressed or Java archive file does not exist, the classpath containing it can still be added to the cache, but ROMClasses from this entry are not stored. If an attempt is made to add a ROMClass to the cache from a directory, but the ROMClass does not exist as a `.class` file, it is not stored in the cache.

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

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

Finding classes

When the JVM finds a class in the cache, it must make more checks than when it stores a class.

When a potential match has been found, if it is a `.class` file on the file system, the time stamps of the `.class` file and the ROMClass stored in the cache are compared. Regardless of the source of the ROMClass (`.jar` or `.class` file), every Java archive and compressed file entry in the calling classpath, up to and including the index at which the ROMClass was “found”, must be checked for updates by obtaining the time stamps. Any update might mean that another version of the class being returned had already been added earlier in the classpath.

Additionally, any classpath entries that are directories might contain `.class` files that “shadow” the potential match that has been found. Class files might be created or deleted in these directories at any point. Therefore, when the classpath is walked and .jar files and compressed files are checked, directory entries are also checked to see whether any `.class` files have been created unexpectedly. This check involves building a string by using the classpath entry, the package names, and the class name, and then looking for the class file. This procedure is expensive if many directories are being used in class paths. Therefore, using .jar files gives better shared classes performance.

Marking classes as stale

When an individual `.class` file is updated, only the class or classes stored from that `.class` file are marked "stale".

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

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

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

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

Classes in the cache that have been loaded from **c**, **d**, and **a** are marked stale. Making a single update to one .jar file might cause many classes in the cache to be marked stale. To avoid massive duplication as classes are updated, stale classes can be marked as not stale, or "redeemed", if it is proved that they are not in fact stale.

Redeeming stale classes

Because classes are marked stale when a class path update occurs, many of the classes marked stale might not have updated. When a class loader stores a class, and in doing so effectively "updates" a stale class, you can "redeem" the stale class if you can prove that it has not in fact changed.

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

AOT code

A single piece of AOT code is associated with a specific method in a specific version of a class in the cache. If new classes are added to the cache as a result of a file system update, new AOT code can be generated for those classes. If a particular class becomes stale, the AOT code associated with that class also becomes stale. If a class is redeemed, the AOT code associated with that class is also redeemed. AOT code is not shared between multiple versions of the same class.

The total amount of AOT code can be limited using **-Xscmaxaot**, and cache space can be reserved for AOT code using **-Xscminaot**.

JIT data

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**, and cache space can be reserved for JIT data using **-Xscminjitdata**.

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 to use the shared classes features in the JVM. The java.net.URLClassLoader shipped with the SDK has been modified to use a SharedClassURLClasspathHelper and any class loaders that extend java.net.URLClassLoader inherit this behavior. Custom class loaders that do not extend URLClassLoader but want to share classes must use the Java Helper API. This topic contains a summary on the different types of Helper API available and how to use them.

The Helper API classes are contained in the com.ibm.oti.shared package. For a detailed description of each helper and how to use it, see the Javadoc information.

com.ibm.oti.shared.Shared
: The Shared class contains static utility methods: getSharedClassHelperFactory() and isSharingEnabled(). If **-Xshareclasses** is specified on the command line and sharing has been successfully initialized, isSharingEnabled() returns true. If sharing is enabled, getSharedClassHelperFactory() returns a com.ibm.oti.shared.SharedClassHelperFactory. The helper factories are singleton factories that manage the Helper APIs. To use the Helper APIs, you must get a Factory.

com.ibm.oti.shared.SharedClassHelperFactory
: SharedClassHelperFactory provides an interface used to create various types of SharedClassHelper for class loaders. Class loaders and SharedClassHelpers have a one-to-one relationship. Any attempts to get a helper for a class loader that already has a different type of helper causes a HelperAlreadyDefinedException.

  Because class loaders and SharedClassHelpers have a one-to-one relationship, calling findHelperForClassLoader() returns a Helper for a given class loader if one exists.

com.ibm.oti.shared.SharedClassHelper
: There are three different types of SharedClassHelper:

  - SharedClassTokenHelper. Use this Helper to store and find classes using a String token generated by the class loader. This Helper is normally used by class loaders that require total control over cache contents.
  - SharedClassURLHelper. Store and find classes using a file system location represented as a URL. For use by class loaders that do not have the concept of a class path and load classes from multiple locations.
  - SharedClassURLClasspathHelper. Store and find classes using a class path of URLs. For use by class loaders that load classes using a URL class path

Compatibility between Helpers is as follows: Classes stored by SharedClassURLHelper can be found using a SharedClassURLClasspathHelper and the opposite also applies. However, classes stored using a SharedClassTokenHelper can be found only by using a SharedClassTokenHelper.

Note: Classes stored using the URL Helpers are updated dynamically by the cache (see "Understanding dynamic updates" on page 355). Classes stored by the SharedClassTokenHelper are not updated by the cache because the Tokens are meaningless Strings, so the Helper has no way of obtaining version information.

You can control the classes a URL Helper finds and stores in the cache using a SharedClassURLFilter. An object implementing this interface can be passed to the SharedClassURLHelper when it is constructed or after it has been created. The filter is then used to decide which classes to find and store in the cache. See "Using the SharedClassHelper API" for more information. For a detailed description of each helper and how to use it, see the Javadoc information.

Using the SharedClassHelper API:

The SharedClassHelper API provides functions to find and store shared classes.

These functions are:

findSharedClass
Called after the class loader has asked its parent for a class, but before it has looked on disk for the class. If findSharedClass returns a class (as a byte[]), pass this class to defineClass(), which defines the class for that JVM and return it as a java.lang.Class object. The byte[] returned by findSharedClass is not the actual class bytes. The effect is that you cannot monitor or manipulate the bytes in the same way as class bytes loaded from a disk. If a class is not returned by findSharedClass, the class is loaded from disk (as in the nonshared case) and then the java.lang.Class defined is passed to storeSharedClass.

storeSharedClass
Called if the class loader has loaded class bytes from disk and has defined them using defineClass. Do not use storeSharedClass to try to store classes that were defined from bytes returned by findSharedClass.

setSharingFilter
Register a filter with the SharedClassHelper. The filter is used to decide which classes are found and stored in the cache. Only one filter can be registered with each SharedClassHelper.

You must resolve how to deal with metadata that cannot be stored. An example is when java.security.CodeSource or java.util.jar.Manifest objects are derived from .jar files. For each .jar file, the best way to deal with metadata that cannot be stored is always to load the first class from the .jar file. Load the class regardless of whether it exists in the cache or not. This load activity initializes the required metadata in the class loader, which can then be cached internally. When a class is then returned by findSharedClass, the function indicates where the class has been loaded from. The result is that the correct cached metadata for that class can be used.

It is not incorrect usage to use storeSharedClass to store classes that were loaded from disk, but which are already in the cache. The cache sees that the class is a

duplicate of an existing class, it is not duplicated, and so the class continues to be shared. However, although it is handled correctly, a class loader that uses only storeSharedClass is less efficient than one that also makes appropriate use of findSharedClass.

Filtering

You can filter which classes are found and stored in the cache by registering an object implementing the SharedClassFilter interface with the SharedClassHelper. Before accessing the cache, the SharedClassHelper functions performs filtering using the registered SharedClassFilter object. For example, you can cache classes inside a particular package only by creating a suitable filter. To define a filter, implement the SharedClassFilter interface, which defines the following methods:

```
boolean acceptStore(String className)
boolean acceptFind(String className)
```

You must return true when you implement these functions so that a class can be found or stored in the cache. Use the supplied parameters as required. Make sure that you implement functions that do not take long to run because they are called for every find and store. Register a filter on a SharedClassHelper using the setSharingFilter(SharedClassFilter filter) function. See the Javadoc for the SharedClassFilter interface for more information.

Applying a global filter

You can apply a SharedClassFilter to all non-bootstrap class loaders that share classes. Specify the com.ibm.oti.shared.SharedClassGlobalFilterClass system property on the command line. For example:

```
-Dcom.ibm.oti.shared.SharedClassGlobalFilterClass=<filter class name>
```

Obtaining information about shared caches:

Use these APIs to obtain information about shared caches.

com.ibm.oti.shared.SharedClassStatistics
: The SharedClassStatistics class provides static utilities that return the total cache size and the amount of free bytes in the active cache.

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 IBM JVMTI extensions for shared class caches, see "Finding shared class caches" on page 399, and "Removing a shared class cache" on page 401.

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.

Verbose output:

The **verbose** suboption of **-Xshareclasses** gives the most concise and simple diagnostic output on cache usage.

See "JVM command-line options" on page 428. Verbose output will typically look like this:

```
>java -Xshareclasses:name=myCache,verbose -Xscmx10k HelloWorld
[-Xshareclasses verbose output enabled]
JVMSHRC158I Successfully created shared class cache "myCache"
JVMSHRC166I Attached to cache "myCache", size=10200 bytes
JVMSHRC096I WARNING: Shared Cache "myCache" is full. Use -Xscmx to set cache size.
Hello
JVMSHRC168I Total shared class bytes read=0. Total bytes stored=9284
```

This output shows that a new cache called myCache was created, which was only 10 kilobytes in size and the cache filled up almost immediately. The message displayed on shut down shows how many bytes were read or stored in the cache.

VerboseIO output:

The verboseIO output is far more detailed, and is used at run time to show classes being stored and found in the cache.

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

```
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 0... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 3... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 17... Failed.
Storing class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 17... Succeeded.
```

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

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

```
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 0... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 3... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 17... Succeeded.
```

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

VerboseHelper output:

You can also obtain diagnostic data from the Java SharedClassHelper API using the **verboseHelper** suboption.

The output is divided into information messages and error messages:

- Information messages are prefixed with:
  `Info for SharedClassHelper id <n>: <message>`
- Error messages are prefixed with:
  `Error for SharedClassHelper id <n>: <message>`

Use the Java Helper API to obtain this output; see "Using the Java Helper API" on page 358.

verboseAOT output:

VerboseAOT provides output when compiled AOT code is being found or stored in the cache.

When a cache is being populated, you might see the following message:

```
Storing AOT code for ROMMethod 0x523B95C0 in shared cache... Succeeded.
```

When a populated cache is being accessed, you might see the following message:

```
Finding AOT code for ROMMethod 0x524EAEB8 in shared cache... Succeeded.
```

AOT code is generated heuristically. You might not see any AOT code generated at all for a small application.

printStats utility:

The **printStats** utility prints summary information about the specified cache to the standard error output. 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
The boundary addresses of the shared memory area containing the classes.

allocPtr
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
The number of bytes of class data in the cache.

AOT bytes
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
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
The number of zip entry cache bytes stored in the cache.

Data bytes
The number of bytes of non-class data stored by the JVM.

Metadata bytes
The number of bytes of data stored to describe the cached classes.

Metadata % used
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
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
The size in bytes of the Class Debug Area that contains data.

Class debug area % used
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
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 358.

 # 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
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 "SharedClassHelper partitions" on page 353. For more information about modification contexts, see "Modification contexts" on page 353.

 # Zip caches
The number of zips that have entry caches stored in the shared cache.

 # Stale classes
The number of classes that have been marked as "potentially stale" by the cache code, because of an operating system update. See "Understanding dynamic updates" on page 355.

% Stale classes
The percentage of classes in the cache that have become stale.

Cache is XXX% full
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*>. This utility lists the cache contents in order, providing as much diagnostic information as possible. Because the output is listed in chronological order, you can interpret it as an "audit trail" of cache updates. Because it is a cache utility, the JVM displays the information about the cache specified or the default cache and then exits.

Each JVM that connects to the cache receives a unique ID. Each entry in the output is preceded by a number indicating the JVM that wrote the data.

Classpaths

```
1: 0x2234FA6C CLASSPATH
        C:\myJVM\sdk\jre\lib\vm.jar
        C:\myJVM\sdk\jre\lib\core.jar
        C:\myJVM\sdk\jre\lib\charsets.jar
        C:\myJVM\sdk\jre\lib\graphics.jar
        C:\myJVM\sdk\jre\lib\security.jar
        C:\myJVM\sdk\jre\lib\ibmpkcs.jar
        C:\myJVM\sdk\jre\lib\ibmorb.jar
        C:\myJVM\sdk\jre\lib\ibmcfw.jar
        C:\myJVM\sdk\jre\lib\ibmorbapi.jar
        C:\myJVM\sdk\jre\lib\ibmjcefw.jar
        C:\myJVM\sdk\jre\lib\ibmjgssprovider.jar
        C:\myJVM\sdk\jre\lib\ibmjsseprovider2.jar
        C:\myJVM\sdk\jre\lib\ibmjaaslm.jar
        C:\myJVM\sdk\jre\lib\ibmjaasactivelm.jar
        C:\myJVM\sdk\jre\lib\ibmcertpathprovider.jar
        C:\myJVM\sdk\jre\lib\server.jar
        C:\myJVM\sdk\jre\lib\xml.jar
```

This output indicates that JVM 1 caused a class path to be stored at address `0x2234FA6C` in the cache. The class path contains 17 entries, which are listed. If the class path is stored using a given partition or modification context, this information is also shown.

ROMClasses

```
1: 0x2234F7DC ROMCLASS: java/lang/Runnable at 0x213684A8
        Index 1 in class path 0x2234FA6C
```

This output indicates that JVM 1 stored a class called java/lang/Runnable in the cache. The metadata about the class is stored at address `0x2234F7DC`, and the class itself is written to address `0x213684A8`. The output also indicates the class path against which the class is stored, and from which index in that class path the class was loaded. In the example, the class path is the same address as the one listed in the Classpath example. If a class is stale, it has `!STALE!` appended to the entry. If the ROMClass is stored using a given partition or modification context, this information is also shown.

AOT methods

```
1: 0x540FBA6A AOT: loadConvert
        for ROMClass java/util/Properties at 0x52345174
```

This output indicates that JVM 1 stored AOT compiled code for the method loadConvert() in `java/util/Properties`. The ROMClass address is the address of the ROMClass that contains the method that was compiled. If an AOT method is stale, it has `!STALE!` appended to the entry.

URLs and Tokens

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

ZipCache

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

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

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.

Debugging problems with shared classes

The following sections describe some of the situations you might encounter with shared classes and also the tools that are available to assist in diagnosing problems.

Using shared classes trace:

Use shared classes trace output only for debugging internal problems or for a detailed trace of activity in the shared classes code.

You enable shared classes trace using the **j9shr** trace component as a suboption of **-Xtrace**. See "Tracing Java applications and the JVM" on page 288 for details. Five levels of trace are provided, level 1 giving essential initialization and runtime information, up to level 5, which is detailed.

Shared classes trace output does not include trace from the port layer functions that deal with memory-mapped files, shared memory, and shared semaphores. It also does not include trace from the Helper API methods. Port layer trace is enabled using the **j9prt** trace component and trace for the Helper API methods is enabled using the **j9jcl** trace component.

Why classes in the cache might not be found or stored:

This quick guide helps you to diagnose why classes might not be being found or stored in the cache as expected.

Why classes might not be found

The class is stale

As explained in "Understanding dynamic updates" on page 355, if a class has been marked as "stale", it is not returned by the cache.

A JVMTI agent is being used without a modification context

If a JVMTI agent is being used without a modification context, classes cannot be found in the cache. The effect is to give the JVMTI agent an opportunity to modify the bytecode when the classes are loaded from disk. For more information, see "Dealing with runtime bytecode modification" on page 351.

The Classpath entry being used is not yet confirmed by the `SharedClassURLClasspathHelper`

Class path entries in the SharedClassURLClasspathHelper must be "confirmed" before classes can be found for these entries. A class path entry is confirmed by having a class stored for that entry. For more information about confirmed entries, see the SharedClassHelper Javadoc information.

Why classes might not be stored

The cache is full

The cache is a finite size, determined when it is created. When it is full, it cannot be expanded. When the **`verbose`** suboption is enabled a message is printed when the cache reaches full capacity, to warn the user. The **`printStats`** utility also displays the occupancy level of the cache, and can be used to query the status of the cache.

The cache is opened read-only

When the **`readonly`** suboption is specified, no data is added to the cache.

The class does not exist on the file system

The class might be sourced from a URL location that is not a file.

The class has been retransformed by JVMTI and `cacheRetransformed` has not been specified

As described in "Dealing with runtime bytecode modification" on page 351, the option **`cacheRetransformed`** must be selected for retransformed classes to be cached.

The class was generated by reflection or Hot Code Replace

These types of classes are never stored in the cache.

Why classes might not be found or stored

The cache is corrupted

In the unlikely event that the cache is corrupted, no classes can be found or stored.

A SecurityManager is being used and the permissions have not been granted to the class loader

SharedClassPermissions must be granted to application class loaders so that they can share classes when a SecurityManager is used. For more information, see "Using SharedClassPermission" on page 168.

Dealing with initialization problems:

Shared classes initialization requires a number of operations to succeed. A failure might have many potential causes, and it is difficult to provide detailed message information following an initialization failure. Some common reasons for failure are listed here.

If you cannot see why initialization has failed from the command-line output, look at level 1 trace for more information regarding the cause of the failure. Review "Operating system limitations" on page 168. A brief summary of potential reasons for failure is provided here.

Writing data into the `javasharedresources` directory

To initialize any cache, data must be written into a `javasharedresources` directory, which is created by the first JVM that needs it.

This directory is /tmp/javasharedresources, and is used only to store small amounts of metadata that identify the semaphore and shared memory areas. .

Problems writing to this directory are the most likely cause of initialization failure. A default cache name is created that includes the username to prevent clashes if different users try to share the same default cache. All shared classes users must also have permissions to write to javasharedresources. The user running the first JVM to share classes on a system must have permission to create the javasharedresources directory.

Caches are created with user-only access by default. Two users cannot share the same cache unless the **-Xshareclasses:groupAccess** command-line option is used when the cache is created. If user A creates a cache using **-Xshareclasses:name=myCache** and user B also tries to run the same command line, a failure occurs. The failure is because user B does not have permissions to access "myCache". Caches can be removed only by the user who created them, even if **-Xshareclasses:groupAccess** is used.

Initializing a persistent cache

The following operations must succeed to initialize a persistent cache:

1) Creating the cache file
Persistent caches are a regular file created on disk. The main reasons for failing to create the file are insufficient disk space and incorrect file permissions.

2) Acquiring file locks
Concurrent access to persistent caches is controlled using operating system file-locking. File locks cannot be obtained if you try to use a cache that is located on a remote networked file system. For example, an NFS or SMB mount. This option is not supported.

3) Memory-mapping the file
The cache file is memory-mapped so that reading and writing to and from it is a fast operation. You cannot memory-map the cache file to a remote networked file system, such as an NFS or SMB mount. This option is not supported. Alternatively, memory-mapping might fail if there is insufficient system memory.

Initializing a non-persistent cache

Non-persistent caches are the default.

The following operations must succeed to initialize a non-persistent cache:

1) Create a shared memory area
The **SHMMAX** operating system environment variable by default is set low. **SHMMAX** limits the size of shared memory segment that can be allocated. If a cache size greater than **SHMMAX** is requested, the JVM attempts to allocate **SHMMAX** and outputs a message indicating that **SHMMAX** should be increased. For this reason, the default cache size is 16 MB.

Before using shared classes on z/OS, check for APARs that must be installed. See "Required APAR for Shared Classes" on page 167. Also, check the operating system environment variables, as detailed in the Chapter 4, "Installing and configuring the SDK," on page 105 section. The requested cache sizes are deliberately rounded to the nearest megabyte.

2) Create a shared semaphore
Shared semaphores are created in the `javasharedresources` directory. You must have write access to this directory.

3) Write metadata
Metadata is written to the `javasharedresources` directory. You must have write access to this directory.

If you are experiencing considerable initialization problems, try a hard reset:

1. Run `java -Xshareclasses:destroyAll` to remove all known memory areas and semaphores. Run this command as root.
2. Delete the `javasharedresources` directory and all of its contents.
3. The memory areas and semaphores created by the JVM might not have been removed using **-Xshareclasses:destroyAll**. This problem is addressed the next time you start the JVM. If the JVM starts and the `javasharedresources` directory does not exist, an automated cleanup is triggered. Any remaining shared memory areas that are shared class caches are removed. Start the JVM with **-Xshareclasses**, using root authority. This action resets the system and forces the JVM to re-create the `javasharedresources` directory.

Dealing with verification problems:

Verification problems (typically seen as java.lang.VerifyErrors) are potentially caused by the cache returning incorrect class bytes.

This problem should not occur under typical usage, but there are two situations in which it could happen:

- The class loader is using a SharedClassTokenHelper and the classes in the cache are out-of-date (dynamic updates are not supported with a SharedClassTokenHelper).
- Runtime bytecode modification is being used that is either not fully predictable in the modifications it does, or it is sharing a cache with another JVM that is doing different (or no) modifications. When you have determined the cause of the problem, destroy the cache, correct the cause of the problem, and try again.

Dealing with cache problems: The following list describes possible cache problems.

Cache is full
A full cache is not a problem; it just means that you have reached the limit of data that you can share. Nothing can be added or removed from that cache and so, if it contains a lot of out-of-date classes or classes that are not being used, you must destroy the cache and create a new one.

Cache is corrupt
In the unlikely event that a cache is corrupt, no classes can be added or read from the cache and a message is output to stderr. If the JVM detects that it is attaching to a corrupted cache, it will attempt to destroy the cache automatically. If the JVM cannot re-create the cache, it will continue to start only if **-Xshareclasses:nonfatal** is specified, otherwise it will exit. If a cache is corrupted during normal operation, all JVMs output the message and are forced to load all subsequent classes locally (not into the cache). The cache is designed to be resistant to crashes, so, if a JVM crash occurs during a cache update, the crash should not cause data to be corrupted.

`Could not create the Java virtual machine` message from utilities
This message does not mean that a failure has occurred. Because the cache

utilities currently use the JVM launcher and they do not start a JVM, this message is always produced by the launcher after a utility has run. Because the JNI return code from the JVM indicates that a JVM did not start, it is an unavoidable message.

-Xscmx is not setting the cache size
You can set the cache size only when the cache is created because the size is fixed. Therefore, **-Xscmx** is ignored unless a new cache is being created. It does not imply that the size of an existing cache can be changed using the parameter.

Class sharing with OSGi ClassLoading framework

Eclipse releases after 3.0 use the OSGi ClassLoading framework, which cannot automatically share classes. A Class Sharing adapter has been written specifically for use with OSGi, which allows OSGi class loaders to access the class cache.

Using the Reliability, Availability, and Serviceability Interface

The JVM Reliability, Availability, and Serviceability Interface (JVMRI) allows an agent to access reliability, availability, and serviceability (RAS) functions by using a structure of pointers to functions.

The JVMRI interface will be deprecated in the near future and replaced by JVMTI extensions.

You can use the JVMRI interface to:

- Determine the trace capability that is present
- Set and intercept trace data
- Produce various dumps
- Inject errors

To use the JVMRI you must be able to build a native library, add the code for JVMRI callbacks (see the subtopics), and interface the code to the JVM through the JNI. This section provides the callback code but does not provide the other programming information.

This chapter describes the JVMRI in:

Preparing to use JVMRI

Trace and dump functions in the JVMRI require the JVM trace and dump libraries to be loaded. These libraries will be loaded by default, but JVMRI will fail with a warning message if you specify **-Xtrace:none** or **-Xdump:none**.

See "Command-line options" on page 417 for more information.

Writing an agent:

This piece of code demonstrates how to write a very simple JVMRI agent.

When an agent is loaded by the JVM, the first thing that gets called is the entry point routine `JVM_OnLoad()`. Therefore, your agent must have a routine called `JVM_OnLoad()`. This routine then must obtain a pointer to the JVMRI function table. This is done by making a call to the `GetEnv()` function.

```
/* jvmri - jvmri agent source file. */

#include "jni.h"
#include "jvmri.h"

DgRasInterface *jvmri_intf = NULL;

JNIEXPORT jint JNICALL
JVM_OnLoad(JavaVM *vm, char *options, void *reserved)
{
    int    rc;
    JNIEnv *env;

    /*
     * Get a pointer to the JNIEnv
     */

    rc = (*vm)->GetEnv(vm, (void **)&env, JNI_VERSION_1_4);
    if (rc != JNI_OK) {
        fprintf(stderr, "RASplugin001 Return code %d obtaining JNIEnv\n", rc);
        fflush(stderr);
        return JNI_ERR;
    }

    /*
     * Get a pointer to the JVMRI function table
     */

    rc = (*vm)->GetEnv(vm, (void **)&jvmri_intf, JVMRAS_VERSION_1_5);
    if (rc != JNI_OK) {
        fprintf(stderr, "RASplugin002 Return code %d obtaining DgRasInterface\n", rc);
        fflush(stderr);
        return JNI_ERR;
    }

    /*
     * Now a pointer to the function table has been obtained we can make calls to any
     * of the functions in that table.
     */

     ....................................................

    return rc;
}
```

Registering a trace listener:

Before you start using the trace listener, you must set the **-Xtrace** option with the relevant **external=tp_spec** information. This action tells the object which tracepoints to listen for.

See “Command-line options” on page 417 for more information.

An agent can register a function that is called back when the JVM makes a trace point. The following example shows a trace listener that only increments a counter each time a trace point is taken.

```
void JNICALL
listener (
    void *env,
    void ** tl,
    const char *moduleName,
    unsigned int traceId,
    const char * format,
    va_list var )
{
    int *counter;

    if (*tl == NULL) {
        fprintf(stderr, "RASplugin100 first tracepoint for thread %p\n", env);
        *tl = (void *)malloc(4);
        counter = (int *)*tl;
        *counter = 0;
    }

    counter = (int *)*tl;

    (*counter)++;

    fprintf(stderr, "Trace point total = %d\n", *counter );
}
```

Add this code to the `JVM_Onload()` function or a function that `JVM_Onload()` calls.

The following example is used to register the trace listener.

```
   /*
    * Register the trace listener
    */

    rc = jvmri_intf->TraceRegister50( env, listener );
    if ( rc != JNI_OK )
    {
        fprintf( stderr, "RASplugin003 Return code %d registering listener\n", rc );
        fflush( stderr );
        return JNI_ERR;
    }
```

You can also do more difficult tasks with a trace listener, including formatting, displaying, and recording trace point information.

Changing trace options:

This example uses the `TraceSet()` function to change the JVM trace setting. It makes the assumption that the options string that is specified with the **-Xrun** option and passed to `JVM_Onload()` is a trace setting.

```
/*
 * If an option was supplied, assume it is a trace setting
 */

 if (options != NULL && strlen(options) > 0) {
     rc = jvmri_intf->TraceSet(env, options);
     if (rc != JNI_OK) {
         fprintf(stderr, "RASplugin004 Return code %d setting trace options\n", rc);
         fflush(stderr);
         return JNI_ERR;
     }
 }
```

To set Maximal tracing for 'j9mm', use the following command when launching the JVM and your agent:

```
java -Xrunjvmri:maximal=j9mm  -Xtrace:external=j9mm App.class
```

Note: Trace must be enabled before the agent can be used. To do this, specify the trace option on the command-line: `-Xtrace:external=j9mm`.

Starting the agent:

To start the agent when the JVM starts up, use the **-Xrun** option. For example if your agent is called jvmri, specify `-Xrunjvmri: <options>` on the command-line.

Building the agent:

You must set some configuration options before you can build a JVMRI agent.

Building the agent on z/OS

To build a JVMRI agent, write a shell script that contains the following entries:

```
SDK_BASE= <sdk directory>
USER_DIR= <user agent's source directory>
c++ -c -g -I$SDK_BASE/include -I$USER_DIR -W "c,float(ieee)"
     -W "c,langlvl(extended)" -W "c,expo,dll" myagent.c
c++ -W "l,dll" -o libmyagent.so myagent.o
chmod 755 libmyagent.so
```

This builds a non-xplink library.

Agent design:

The agent must reference the header files jni.h and jvmri.h, which are shipped with the SDK and are in the sdk\include subdirectory.

To start the agent, use the **-Xrun** command-line option. The JVM parses the **-Xrunlibrary_name[:options]** switch and loads library_name if it exists. A check for an entry point that is called `JVM_OnLoad` is then made. If the entry point exists, it is called to allow the library to initialize. This processing occurs after the initialization of all JVM subcomponents. The agent can then call the functions that have been initialized, by using the JVMRI table.

JVMRI functions

At startup, the JVM initializes JVMRI. You access the JVMRI functions with the JNI `GetEnv()` routine to obtain an interface pointer.

For example:

```
JNIEXPORT jint JNICALL
JVM_OnLoad(JavaVM *vm, char *options, void *reserved)
{
 DgRasInterface *ri;
   ......
    (*vm)->GetEnv(vm, (void **)&ri, JVMRAS_VERSION_1_5)

    rc = jvmras_intf->TraceRegister50(env, listener);
   ......
}
```

API calls provided by JVMRI

The JVMRI functions are defined in a header file `jvmri.h`, which is supplied in the `sdk/include` directory. Note that all calls must be made using a valid JNIEnv pointer as the first parameter.

The TraceRegister and TraceDeregister functions are deprecated. Use TraceRegister50 and TraceDeregister50.

CreateThread:

```
int CreateThread( JNIEnv *env, void JNICALL (*startFunc)(void*),
      void *args, int GCSuspend)
```

Description

Creates a thread. A thread can be created only after the JVM has been initialized. However, calls to CreateThread can be made also before initialization; the threads are created by a callback function after initialization.

Parameters

- A valid pointer to a JNIEnv.
- Pointer to start function for the new thread.
- Pointer to argument that is to be passed to start function.
- GCSuspend parameter is ignored.

Returns

JNI Return code `JNI_OK` if thread creation is successful; otherwise, `JNI_ERR`.

DumpDeregister:

```
int DumpDeregister(JNIEnv *env,  int (JNICALL *func)(JNIEnv *env2,
      void **threadLocal, int reason))
```

Description

De-registers a dump call back function that was previously registered by a call to DumpRegister.

Parameters

- A valid pointer to a JNIEnv.
- Function pointer to a previously registered dump function.

Returns

JNI return codes `JNI_OK` and `JNI_EINVAL`.

DumpRegister:

```
int DumpRegister(JNIEnv *env, int (JNICALL *func)(JNIEnv *env2,
      void **threadLocal, int reason))
```

Description

Registers a function that is called back when the JVM is about to generate a JavaCore file.

Parameters

- A valid pointer to a JNIEnv.
- Function pointer to dump function to register.

Returns

JNI return codes `JNI_OK` and `JNI_ENOMEM`.

DynamicVerbosegc:

```
void JNICALL *DynamicVerbosegc (JNIEnv *env, int vgc_switch,
             int vgccon, char* file_path, int number_of_files,
             int number_of_cycles);
```

Description

Not supported. Displays the message "not supported".

Parameters

- A valid pointer to a JNIEnv.
- Integer that indicates the direction of switch (JNI_TRUE = on, JNI_FALSE = off)
- Integer that indicates the level of verbosegc (0 = **-verbose:gc**, 1 = **-verbose:Xgccon**)
- Pointer to string that indicates file name for file redirection
- Integer that indicates the number of files for redirection
- Integer that indicates the number of cycles of verbose:gc per file

Returns

None.

GenerateHeapdump:

```
int GenerateHeapdump( JNIEnv *env )
```

Description

Generates a Heapdump file.

Parameters

- A valid pointer to a JNIEnv.

Returns

JNI Return code `JNI_OK` if running dump is successful; otherwise, `JNI_ERR`.

GenerateJavacore:

```
int GenerateJavacore( JNIEnv *env )
```

Description

Generates a Javacore file.

Parameters

- A valid pointer to a JNIEnv.

Returns

JNI Return code `JNI_OK` if running dump is successful; otherwise, `JNI_ERR`.

GetComponentDataArea:

```
int GetComponentDataArea( JNIEnv *env, char *componentName,
       void **dataArea, int *dataSize )
```

Description

Not supported. Displays the message `no data area for <requested component>`.

Parameters

- A valid pointer to a JNIEnv.
- Component name.
- Pointer to the component data area.
- Size of the data area.

Returns

JNI_ERR

GetRasInfo:

```
int GetRasInfo(JNIEnv * env,
     RasInfo * info_ptr)
```

Description

This function fills in the supplied RasInfo structure, based on the request type that is initialized in the RasInfo structure. (See details of the RasInfo structure in "RasInfo structure" on page 382.

Parameters

- A valid pointer to a JNIEnv. This parameter is reserved for future use.
- Pointer to a RasInfo structure. This should have the **type** field initialized to a supported request.

Returns

JNI Return codes `JNI_OK`, `JNI_EINVAL` and `JNI_ENOMEM`.

InitiateSystemDump:

```
int JNICALL InitiateSystemDump( JNIEnv *env )
```

Description

Initiates a system dump. The dumps and the output that are produced depend on the settings for JAVA_DUMP_OPTS and JAVA_DUMP_TOOL and on the support that is offered by each platform.

Parameters

- A valid pointer to a JNIEnv.

Returns

JNI Return code `JNI_OK` if dump initiation is successful; otherwise, `JNI_ERR`. If a specific platform does not support a system-initiated dump, `JNI_EINVAL` is returned.

InjectOutOfMemory:

```
int InjectOutOfMemory( JNIEnv *env )
```

Description

Causes native memory allocations made after this call to fail. This function is intended to simulate exhaustion of memory allocated by the operating system.

Parameters

- A valid pointer to a JNIEnv.

Returns

JNI_OK if the native allocation function is successfully swapped for the JVMRI function that always returns NULL, JNI_ERR if the swap is unsuccessful.

InjectSigSegv:

```
int InjectSigsegv( JNIEnv *env )
```

Description

Raises a SIGSEGV exception, or the equivalent for your platform.

Parameters

- A valid pointer to a JNIEnv.

Returns

JNI_ERR

NotifySignal:

```
void NotifySignal(JNIEnv *env, int signal)
```

Description

Raises a signal in the JVM.

Parameters

- A valid pointer to a JNIEnv. This parameter is reserved for future use.
- Signal number to raise.

Returns

Nothing.

ReleaseRasInfo:

```
int ReleaseRasInfo(JNIEnv * env,
   RasInfo * info_ptr)
```

Description

This function frees any areas to which the RasInfo structure might point after a successful `GetRasInfo` call. The request interface never returns pointers to 'live' JVM control blocks or variables.

Parameters

- A valid pointer to a JNIEnv. This parameter is reserved for future use.
- Pointer to a RasInfo structure. This should have previously been set up by a call to GetRasInfo. An error occurs if the **type** field has not been initialized to a supported request. (See details of the RasInfo structure in "RasInfo structure" on page 382.)

Returns

JNI Return codes `JNI_OK` or `JNI_EINVAL`.

RunDumpRoutine:

```
int RunDumpRoutine( JNIEnv *env, int componentID, int level, void (*printrtn)
 (void *env, const char *tagName, const char *fmt, ...) )
```

Description

Not supported. Displays the message `?not supported?`.

Parameters

- A valid pointer to a JNIEnv.
- Id of component to dump.
- Detail level of dump.
- Print routine to which dump output is directed.

Returns

`JNI_ERR`

SetOutOfMemoryHook:

```
int SetOutOfMemoryHook( JNIEnv *env, void (*rasOutOfMemoryHook)
        (void) )
```

Description

Registers a callback function for an out-of-memory condition.

Parameters

- A valid pointer to a JNIEnv.
- Pointer to callback function.

Returns
JNI Return code JNI_OK if table is successfully updated; otherwise, JNI_ERR.

TraceDeregister:

```
int TraceDeregister(JNIEnv *env,  void (JNICALL *func)(JNIEnv *env2,
       void **threadLocal, int traceId, const char *
       format, va_list varargs))
```

Description
Deregisters an external trace listener.

Important: This function is now deprecated. Use "TraceDeregister50."

Parameters

- A valid pointer to a JNIEnv.
- Function pointer to a previously-registered trace function.

Returns
JNI Return code JNI_OK or JNI_EINVAL.

TraceDeregister50:

```
int TraceDeregister50 (
    JNIEnv *env,
    void ( JNICALL *func ) (
        JNIEnv *env2,
        void **threadLocal,
        const char *moduleName,
        int traceId,
        const char *format,
        va_list varargs
        )
    )
```

Description
Deregisters an external trace listener.

Parameters

- A valid pointer to a JNIEnv.
- Function pointer to a previously-registered trace function.

Returns
JNI Return code JNI_OK or JNI_EINVAL.

TraceRegister:

```
int TraceRegister(JNIEnv *env, void (JNICALL *func)(JNIEnv *env2,
      void **threadLocal, int traceId, const char * format,
       va_list var))
```

Description
Registers a trace listener.

Important: This function is now deprecated. Use "TraceRegister50" on page 381.

Parameters

- A valid pointer to a JNIEnv.
- Function pointer to trace function to register.

Returns
JNI Return code JNI_OK or JNI_ENOMEM.

TraceRegister50:

```
int TraceRegister50 (
    JNIEnv *env,
    void ( JNICALL *func ) (
        JNIEnv *env2,
        void **threadLocal,
        const char *moduleName,
        int traceId,
        const char *format,
        va_list varargs
        )
    )
```

Description

Registers a trace listener.

Parameters

- A valid pointer to a JNIEnv.
- Function pointer to trace function to register.

Returns

JNI Return code `JNI_OK` or `JNI_ENOMEM`.

TraceResume:

```
void TraceResume(JNIEnv *env)
```

Description

Resumes tracing.

Parameters

- A valid pointer to a JNIEnv. If MULTI_JVM; otherwise, it can be NULL.

Returns

Nothing.

TraceResumeThis:

```
void TraceResumeThis(JNIEnv *env);
```

Description

Resume tracing from the current thread. This action decrements the *resumecount* for this thread. When it reaches zero (or less) the thread *starts* tracing (see "Tracing Java applications and the JVM" on page 288).

Parameters

- A valid pointer to a JNIEnv.

Returns

None.

TraceSet:

```
int TraceSet(JNIEnv *env, const char *cmd)
```

Description

Sets the trace configuration options. This call parses only the first valid trace command passed to it, but can be called multiple times. Hence, to achieve the equivalent of setting `-Xtrace:maximal=j9mm,iprint=j9shr`, you call TraceSet twice, once with the cmd parameter `maximal=j9mm` and once with `iprint=j9shr`.

Parameters

- A valid pointer to a JNIEnv.
- Trace configuration command.

Returns
JNI Return code JNI_OK, JNI_ERR, JNI_ENOMEM, JNI_EXIST and JNI_EINVAL.

TraceSnap:

```
void TraceSnap(JNIEnv *env, char *buffer)
```

Description
Takes a snapshot of the current trace buffers.

Parameters

- A valid pointer to a JNIEnv; if set to NULL, current Execenv is used.
- The second parameter is no longer used, but still exists to prevent changing the function interface. It can safely be set to NULL.

Returns
Nothing

TraceSuspend:

```
void TraceSuspend(JNIEnv *env)
```

Description
Suspends tracing.

Parameters

- A valid pointer to a JNIEnv; if MULTI_JVM; otherwise, it can be NULL.

Returns
Nothing.

TraceSuspendThis:

```
void TraceSuspendThis(JNIEnv *env);
```

Description
Suspend tracing from the current thread. This action decrements the *suspendcount* for this thread. When it reaches zero (or less) the thread *stops* tracing (see "Tracing Java applications and the JVM" on page 288).

Parameters

- A valid pointer to a JNIEnv.

Returns
None.

RasInfo structure

The RasInfo structure that is used by GetRasInfo() takes the following form. (Fields that are initialized by GetRasInfo are underscored):

```
typedef struct RasInfo {
     int type;
     union {
  struct {
      int   number;
      char  **names;
  } query;
  struct {
      int   number;
      char  **names;
  } trace_components;
  struct {
      char  *name
      int   first;
      int   last;
```

```
    unsigned char  *bitMap;
  } trace_component;
    } info;
 } RasInfo;
```

RasInfo request types

The following request types are supported:

RASINFO_TYPES
Returns the *number* of request types that are supported and an array of pointers to their names in the enumerated sequence. The names are in code page ISO8859-1.

RASINFO_TRACE_COMPONENTS
Returns the *number* of components that can be enabled for trace and an array of pointers to their *names* in the enumerated sequence. The names are in code page ISO8859-1.

RASINFO_TRACE_COMPONENT
Returns the *first* and *last* tracepoint ids for the component *name* (code page ISO8859-1) and a *bitmap* of those tracepoints, where a 1 signifies that the tracepoint is in the build. The *bitmap* is big endian (tracepoint ID *first* is the most significant bit in the first byte) and is of length ((last-first)+7)/8 bytes.

Intercepting trace data

To receive trace information from the JVM, you can register a trace listener using JVMRI. In addition, you must specify the option **-Xtrace:external=<option>** to route trace information to an external trace listener.

The -Xtrace:external=<option>: The format of this property is:

`-Xtrace:external=[[!]tracepoint_specification[,...]]`

This system property controls what is traced. Multiple statements are allowed and their effect is cumulative.

The *tracepoint_specification* is as follows:

*Component***[(***Class***[,...])]**
Where ***component*** is the JVM subcomponent or **all**. If no component is specified, **all** is assumed.

class is the tracepoint type or **all**. If class is not specified, **all** is assumed.

TPID(*tracepoint_id***[,...])**
Where *tracepoint_id* is the hexadecimal global tracepoint identifier.

If no qualifier parameters are entered, all tracepoints are enabled; that is, the equivalent of specifying **all**.

The ! (exclamation mark) is a logical not. It allows complex tracepoint selection.

Calling external trace: If an external trace routine has been registered and a tracepoint has been enabled for external trace, it is called with the following parameters:

env
Pointer to the JNIEnv for the current thread.

traceid
Trace identifier

format
A zero-terminated string that describes the format of the variable argument list that follows. The possible values for each character position are:

0x01 One character
0x02 Short
0x04 Int
0x08 Double or long long
0xfe Pointer to java/lang/String object
0xff ASCII string pointer (can be NULL)
0x00 End of format string

If the format pointer is NULL, no trace data follows.

varargs
A va_list of zero or more arguments as defined in **format** argument.

Formatting

You can use `J9TraceFormat.dat` to format JVM-generated tracepoints that are captured by the agent. `J9TraceFormat.dat` is shipped with the SDK.

`J9TraceFormat.dat` consists of a flat ASCII or EBCDIC file of the following format:

```
5.0
j9vm 0 1 1 N Trc_VM_VMInitStages_Event1 " Trace engine initialized for module j9vm"
j9vm 2 1 1 N Trc_VM_CreateRAMClassFromROMClass_Entry " >Create RAM class from ROM class %p in class loader %p"
j9vm 4 1 1 N Trc_VM_CreateRAMClassFromROMClass_Exit " j9vm 4 1 1 N Trc_VM_CreateRAMClassFromROMClass_Exit "
```

The first line contains the version number of the format file. A new version number reflects changes to the layout of this file.

The format of each tracepoint entry is as follows:

<component> <t> <o> <l> <e> <symbol> <template>

where:

- *<component>* is the internal JVM component name.
- *<t>* is the tracepoint type (0 through 11).
- *<o>* is the overhead (0 through 10).
- *<l>* is the level of the tracepoint (0 through 9, or **-** if the tracepoint is obsolete).
- <e> is the explicit setting flag (Y/N).
- *<symbol>* is the name of the tracepoint.
- *<template>* is a template that is used to format the entry. The template consists of the text that appears in double quotation marks (").

Tracepoint types are as follows:

Type 0
Event

Type 1
Exception

Type 2
Entry

Type 4
Exit

Type 5
Exit-with-Exception

Type 6
Mem

Any other type is reserved for development use; you should not find any other type on a release version of IBM Java.

Note: This condition is subject to change without notice.

The version number is different for each version.

Using the HPROF Profiler

HPROF is a demonstration profiler shipped with the IBM SDK that uses the JVMTI to collect and record information about Java execution. You can use **HPROF** to work out which parts of a program are using the most memory or processor time.

Note: For analyzing memory usage, you should use IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer, which is a newer tool. For more information about this tool, see "Using the IBM Monitoring and Diagnostic Tools for Java" on page 219.

To improve the efficiency of your applications, you must know which parts of the code are using large amounts of memory and processor resources. HPROF is an example JVMTI agent and is started using the following syntax:

```
java -Xrunhprof[:<option>=<value>,...] <classname>
```

When you run Java with HPROF, a file is created when the program ends. This file is placed in the current working directory and is called `java.hprof.txt` (`java.hprof` if binary format is used) unless a different file name has been given. This file contains a number of different sections, but the exact format and content depend on the selected options.

If you need more information about HPROF than is contained in this section, see http://java.sun.com/developer/technicalArticles/Programming/HPROF.html.

The command `java -Xrunhprof:help` shows the options available:

heap=dump|sites|all
This option helps in the analysis of memory usage. It tells HPROF to generate stack traces, from which you can see where memory was allocated. If you use the **heap=dump** option, you get a dump of all live objects in the heap. With **heap=sites**, you get a sorted list of sites with the most heavily allocated objects at the start. The default value **all** gives both types of output.

cpu=samples|times|old
The **cpu** option provides information that is useful in determining where the processor spends most of its time. If **cpu** is set to **samples**, the JVM pauses execution and identifies which method call is active. If the sampling rate is high enough, you get a good picture of where your program spends most of its time. If **cpu** is set to **time**, you receive precise measurements of how many times each method was called and how long each execution took. Although this option is more accurate, it slows down the program. If **cpu** is set to **old**, the profiling data is produced in the old HPROF format.

interval=y|n
The interval option applies only to **cpu=samples** and controls the time that the sampling thread sleeps between samples of the thread stacks.

monitor=y|n
The **monitor** option can help you understand how synchronization affects the performance of your application. Monitors implement thread synchronization. Getting information about monitors can tell you how much time different threads are spending when trying to access resources that are already locked. HPROF also gives you a snapshot of the monitors in use. This information is useful for detecting deadlocks.

format=a|b
The default for the output file is ASCII format. Set **format** to 'b' if you want to specify a binary format, which is required for some utilities like the Heap Analysis Tool.

file=<filename>
Use the **file** option to change the name of the output file. The default name for an ASCII file is `java.hprof.txt`. The default name for a binary file is `java.hprof`.

force=y|n
Typically, the default (**force=y**) overwrites any existing information in the output file. So, if you have multiple JVMs running with HPROF enabled, use **force=n**, which appends additional characters to the output file name as needed.

net=<host>:<port>
To send the output over the network rather than to a local file, use the **net** option.

depth=<size>
The **depth** option indicates the number of method frames to display in a stack trace. The default is 4.

thread=y|n
If you set the **thread** option to **y**, the thread id is printed next to each trace. This option is useful if you cannot see which thread is associated with which trace. This type of problem might occur in a multi-threaded application.

doe=y|n
The default behavior is to collect profile information when an application exits. To collect the profiling data during execution, set **doe** (dump on exit) to **n**.

msa=y|n
This feature is unsupported on IBM SDK platforms.

cutoff=<value>
Many sample entries are produced for a small percentage of the total execution time. By default, HPROF includes all execution paths that represent at least 0.0001 percent of the time spent by the processor. You can increase or decrease that cutoff point using this option. For example, to eliminate all entries that represent less than one-fourth of one percent of the total execution time, you specify **cutoff=0.0025**.

verbose=y|n
This option generates a message when dumps are taken. The default is **y**.

lineno=y|n
Each frame typically includes the line number that was processed, but you can

use this option to suppress the line numbers from the output listing. If enabled, each frame contains the text `Unknown line` instead of the line number.

```
TRACE 1056:
java/util/Locale.toUpperCase(Locale.java:Unknown line)
java/util/Locale.<init>(Locale.java:Unknown line)
java/util/Locale.<clinit>(Locale.java:Unknown line)
sun/io/CharacterEncoding.aliasName(CharacterEncoding.java:Unknown line)
```

Explanation of the HPROF output file

The first section of the file contains general header information such as an explanation of the options, copyright, and disclaimers. A summary of each thread follows.

You can see the output after using HPROF with a simple program, shown as follows. This test program creates and runs two threads for a short time. From the output, you can see that the two threads called `apples` and then `oranges` were created after the system-generated `main` thread. Both threads end before the `main` thread. For each thread its address, identifier, name, and thread group name are displayed. You can see the order in which threads start and finish.

```
THREAD START (obj=11199050, id = 1, name="Signal dispatcher", group="system")
THREAD START (obj=111a2120, id = 2, name="Reference Handler", group="system")
THREAD START (obj=111ad910, id = 3, name="Finalizer", group="system")
THREAD START (obj=8b87a0, id = 4, name="main", group="main")
THREAD END (id = 4)
THREAD START (obj=11262d18, id = 5, name="Thread-0", group="main")
THREAD START (obj=112e9250, id = 6, name="apples", group="main")
THREAD START (obj=112e9998, id = 7, name="oranges", group="main")
THREAD END (id = 6)
THREAD END (id = 7)
THREAD END (id = 5)
```

The trace output section contains regular stack trace information. The depth of each trace can be set and each trace has a unique ID:

```
TRACE 5:
 java/util/Locale.toLowerCase(Locale.java:1188)
 java/util/Locale.convertOldISOCodes(Locale.java:1226)
 java/util/Locale.<init>(Locale.java:273)
 java/util/Locale.<clinit>(Locale.java:200)
```

A trace contains a number of frames, and each frame contains the class name, method name, file name, and line number. In the previous example, you can see that line number 1188 of `Locale.java` (which is in the toLowerCase method) has been called from the convertOldISOCodes() function in the same class. These traces are useful in following the execution path of your program. If you set the monitor option, a monitor dump is produced that looks like this example:

```
MONITOR DUMP BEGIN
    THREAD 8, trace 1, status: R
    THREAD 4, trace 5, status: CW
    THREAD 2, trace 6, status: CW
    THREAD 1, trace 1, status: R
    MONITOR java/lang/ref/Reference$Lock(811bd50) unowned
 waiting to be notified: thread 2
    MONITOR java/lang/ref/ReferenceQueue$Lock(8134710) unowned
 waiting to be notified: thread 4
    RAW MONITOR "_hprof_dump_lock"(0x806d7d0)
 owner: thread 8, entry count: 1
    RAW MONITOR "Monitor Cache lock"(0x8058c50)
 owner: thread 8, entry count: 1
    RAW MONITOR "Monitor Registry lock"(0x8058d10)
 owner: thread 8, entry count: 1
    RAW MONITOR "Thread queue lock"(0x8058bc8)
```

```
 owner: thread 8, entry count: 1
MONITOR DUMP END
MONITOR TIME BEGIN (total = 0 ms) Thu Aug 29 16:41:59 2002
MONITOR TIME END
```

The first part of the monitor dump contains a list of threads, including the trace entry that identifies the code the thread executed. There is also a thread status for each thread where:

- R — Runnable (The thread is able to run when given the chance)
- S — Suspended (The thread has been suspended by another thread)
- CW — Condition Wait (The thread is waiting)
- MW — Monitor Wait (The monitor is waiting)

Next is a list of monitors along with their owners and an indication of whether there are any threads waiting on them.

The Heapdump is the next section. This information contains a list of the different areas of memory, and shows how they are allocated:

```
CLS 1123edb0 (name=java/lang/StringBuffer, trace=1318)
 super  111504e8
 constant[25] 8abd48
 constant[32] 1123edb0
 constant[33] 111504e8
 constant[34] 8aad38
 constant[115] 1118cdc8
CLS 111ecff8 (name=java/util/Locale, trace=1130)
 super  111504e8
 constant[2] 1117a5b0
 constant[17] 1124d600
 constant[24] 111fc338
 constant[26] 8abd48
 constant[30] 111fc2d0
 constant[34] 111fc3a0
 constant[59] 111ecff8
 constant[74] 111504e8
 constant[102] 1124d668
 ...
CLS 111504e8 (name=java/lang/Object, trace=1)
 constant[18] 111504e8
```

CLS tells you that memory is being allocated for a class. The hexadecimal number following it is the address where that memory is allocated.

Next is the class name followed by a trace reference. Use this information to cross-reference the trace output and see when the class is called. If you refer to that particular trace, you can get the line number of the instruction that led to the creation of this object. The addresses of the constants in this class are also displayed and, in the previous example, the address of the class definition for the superclass. Both classes are a child of the same superclass (with address `11504e8`). Looking further through the output, you can see this class definition and name. It is the Object class (a class that every class inherits from). The JVM loads the entire superclass hierarchy before it can use a subclass. Thus, class definitions for all superclasses are always present. There are also entries for Objects (`OBJ`) and Arrays (`ARR`):

```
OBJ 111a9e78 (sz=60, trace=1, class=java/lang/Thread@8b0c38)
 name  111afbf8
 group  111af978
 contextClassLoader 1128fa50
 inheritedAccessControlContext 111aa2f0
```

```
 threadLocals 111bea08
 inheritableThreadLocals 111bea08
ARR 8bb978 (sz=4, trace=2, nelems=0, elem type=java/io/ObjectStreamField@8bac80)
```

If you set the **heap** option to **sites** or **all** , you get a list of each area of storage allocated by your code. The parameter **all** combines **dump** and **sites**. This list is ordered starting with the sites that allocate the most memory:

```
SITES BEGIN (ordered by live bytes) Tue Feb 06 10:54:46 2007
          percent          live          alloc'ed  stack class
 rank   self  accum     bytes objs     bytes  objs trace name
    1 20.36% 20.36%    190060   16    190060    16 300000 byte[]
    2 14.92% 35.28%    139260 1059    139260  1059 300000 char[]
    3  5.27% 40.56%     49192   15     49192    15 300055 byte[]
    4  5.26% 45.82%     49112   14     49112    14 300066 byte[]
    5  4.32% 50.14%     40308 1226     40308  1226 300000 java.lang.String
    6  1.62% 51.75%     15092  438     15092   438 300000 java.util.HashMap$Entry
    7  0.79% 52.55%      7392   14      7392    14 300065 byte[]
    8  0.47% 53.01%      4360   16      4360    16 300016 char[]
    9  0.47% 53.48%      4352   34      4352    34 300032 char[]
   10  0.43% 53.90%      3968   32      3968    32 300028 char[]
   11  0.40% 54.30%      3716    8      3716     8 300000 java.util.HashMap$Entry[]
   12  0.40% 54.70%      3708   11      3708    11 300000 int[]
   13  0.31% 55.01%      2860   16      2860    16 300000 java.lang.Object[]
   14  0.28% 55.29%      2644   65      2644    65 300000 java.util.Hashtable$Entry
   15  0.28% 55.57%      2640   15      2640    15 300069 char[]
   16  0.27% 55.84%      2476   17      2476    17 300000 java.util.Hashtable$Entry[]
   17  0.25% 56.08%      2312   16      2312    16 300013 char[]
   18  0.25% 56.33%      2312   16      2312    16 300015 char[]
   19  0.24% 56.57%      2224   10      2224    10 300000 java.lang.Class
```

In this example, Trace 300055 allocated 5.27% of the total allocated memory. This percentage works out to be 49192 bytes.

The **cpu** option gives profiling information about the processor. If **cpu** is set to samples, the output contains the results of periodic samples taken during execution of the code. At each sample, the code path being processed is recorded, and a report is produced similar to:

```
CPU SAMPLES BEGIN (total = 714) Fri Aug 30 15:37:16 2002
rank   self  accum   count trace method
1 76.28% 76.28% 501 77 MyThread2.bigMethod
2 6.92% 83.20% 47 75 MyThread2.smallMethod
...
CPU SAMPLES END
```

You can see that the bigMethod() was responsible for 76.28% of the processor execution time and was being run 501 times out of the 714 samples. If you use the trace IDs, you can see the exact route that led to this method being called.

Using the JVMTI

JVMTI is a two-way interface that allows communication between the JVM and a native agent. It replaces the JVMDI and JVMPI interfaces.

JVMTI allows third parties to develop debugging, profiling, and monitoring tools for the JVM. The interface contains mechanisms for the agent to notify the JVM about the kinds of information it requires. The interface also provides a means of receiving the relevant notifications. Several agents can be attached to a JVM at any one time. A number of tools are based on this interface, including IBM Monitoring and Diagnostic Tools for Java - Health Center. For more information about IBM Monitoring and Diagnostic Tools for Java, see “Using the IBM Monitoring and Diagnostic Tools for Java” on page 219.

JVMTI agents can be loaded at startup using short or long forms of the command-line option:

```
-agentlib:<agent-lib-name>=<options>
```

or

```
-agentpath:<path-to-agent>=<options>
```

For example:

```
-agentlib:hprof=<options>
```

assumes that a folder containing `hprof.dll` is on the library path, or

```
-agentpath:C:\sdk\jre\bin\hprof.dll=<options>
```

For more information about JVMTI, see http://download.oracle.com/javase/7/docs/technotes/guides/jvmti/.

For advice on porting JVMPI-based profilers to JVMTI, see http://www.oracle.com/technetwork/articles/javase/jvmpitransition-138768.html.

For a guide about writing a JVMTI agent, see http://www.oracle.com/technetwork/articles/javase/jvmti-136367.html.

IBM JVMTI extensions

The IBM SDK provides extensions to the JVMTI. The sample shows you how to write a simple JVMTI agent that uses these extensions.

The IBM SDK extensions to JVMTI allow a JVMTI agent to do the following tasks:

- Modify a dump.
- Modify a trace.
- Modify the logging configuration of the JVM.
- Start a JVM dump.
- Query the JRE native memory use.
- Find and remove shared class caches.
- Subscribe to and unsubscribe from verbose garbage collection logging.

The definitions that you need when you write a JVMTI agent are provided in the header files `jvmti.h` and `ibmjvmti.h`. These files are in `sdk/include`.

The sample JVMTI agent consists of two functions:

1. Agent_OnLoad()
2. DumpStartCallback()

Agent_OnLoad()

This function is called by the JVM when the agent is loaded at JVM startup, which allows the JVMTI agent to modify JVM behavior before initialization is complete. The sample agent obtains access to the JVMTI interface by using the JNI Invocation API function GetEnv(). The agent calls the APIs GetExtensionEvents() and GetExtensionFunctions() to find the JVMTI extensions that are supported by the JVM. These APIs provide access to the list of extensions available in the jvmtiExtensionEventInfo and jvmtiExtensionFunctionInfo structures. The sample uses an extension event and an extension function in the following way:

The sample JVMTI agent searches for the extension event VmDumpStart in the list of jvmtiExtensionEventInfo structures, by using the identifier COM_IBM_VM_DUMP_START provided in ibmjvmti.h. When the event is found, the JVMTI agent calls the JVMTI interface SetExtensionEventCallback() to enable the event, providing a function DumpStartCallback() that is called when the event is triggered.

Next, the sample JVMTI agent searches for the extension function SetVMDump in the list of jvmtiExtensionFunctionInfo structures, by using the identifier COM_IBM_SET_VM_DUMP provided in ibmjvmti.h. The JVMTI agent calls the function by using the jvmtiExtensionFunction pointer to set a JVM dump option java:events=thrstart. This option requests the JVM to trigger a Javadump every time a VM thread is started.

DumpStartCallback()

This callback function issues a message when the associated extension event is called. In the sample code, DumpStartCallback() is used when the VmDumpStart event is triggered.

Compiling and running the sample JVMTI agent

Use this command to build the sample JVMTI agent on Linux:

```
gcc -I<SDK_path>/include -o libtiSample.so -shared tiSample.c
```

where <SDK_path> is the path to your SDK installation.

To run the sample JVMTI agent, use the command:

```
java -agentlib:tiSample -version
```

When the sample JVMTI agent loads, messages are generated. When the JVMTI agent initiates a Javadump, the message JVMDUMP010 is issued.

Sample JVMTI agent:

A sample JVMTI agent, written in C/C++, using the IBM JVMTI extensions.

```
/*
 * tiSample.c
 *
 * Sample JVMTI agent to demonstrate the IBM JVMTI dump extensions
 */

#include "jvmti.h"
#include "ibmjvmti.h"

/* Forward declarations for JVMTI callback functions */
void JNICALL VMInitCallback(jvmtiEnv *jvmti_env, JNIEnv* jni_env, jthread thread);
void JNICALL DumpStartCallback(jvmtiEnv *jvmti_env, char* label, char* event, char* detail, ...);


/*
 * Agent_Onload()
 *
 * JVMTI agent initialisation function, invoked as agent is loaded by the JVM
 */
JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *jvm, char *options, void *reserved) {

  jvmtiEnv *jvmti = NULL;
 jvmtiError rc;
```

```
 jint extensionEventCount = 0;
 jvmtiExtensionEventInfo *extensionEvents = NULL;
 jint extensionFunctionCount = 0;
 jvmtiExtensionFunctionInfo *extensionFunctions = NULL;
 int i = 0, j = 0;

  printf("tiSample: Loading JVMTI sample agent\n");

    /* Get access to JVMTI */
    (*jvm)->GetEnv(jvm, (void **)&jvmti, JVMTI_VERSION_1_0);

    /* Look up all the JVMTI extension events and functions */
   (*jvmti)->GetExtensionEvents(jvmti, &extensionEventCount, &extensionEvents);
   (*jvmti)->GetExtensionFunctions(jvmti, &extensionFunctionCount, &extensionFunctions);

    printf("tiSample: Found %i JVMTI extension events, %i extension functions\n", extensionEventCount,
extensionFunctionCount);

   /* Find the JVMTI extension event we want */
   while (i++ < extensionEvenCount) {

        if (strcmp(extensionEvents->id, COM_IBM_VM_DUMP_START) == 0) {
            /* Found the dump start extension event, now set up a callback for it */
            rc = (*jvmti)->SetExtensionEventCallback(jvmti, extensionEvents->extension_event_index,
&DumpStartCallback);
            printf("tiSample: Setting JVMTI event callback %s, rc=%i\n", COM_IBM_VM_DUMP_START, rc);
            break;
        }
        extensionEvents++; /* move on to the next extension event */
    }

    /* Find the JVMTI extension function we want */
    while (j++ < extensionFunctionCount) {
        jvmtiExtensionFunction function = extensionFunctions->func;

        if (strcmp(extensionFunctions->id, COM_IBM_SET_VM_DUMP) == 0) {
            /* Found the set dump extension function, now set a dump option to generate javadumps on
     thread starts */
            rc = function(jvmti, "java:events=thrstart");
            printf("tiSample: Calling JVMTI extension %s, rc=%i\n", COM_IBM_SET_VM_DUMP, rc);
            break;
        }
        extensionFunctions++; /* move on to the next extension function */
     }

     return JNI_OK;

}


/*
 * DumpStartCallback()
 * JVMTI callback for dump start event (IBM JVMTI extension)  */
void JNICALL
DumpStartCallback(jvmtiEnv *jvmti_env, char* label, char* event, char* detail, ...) {
   printf("tiSample: Received JVMTI event callback, for event %s\n", event);
}
```

IBM JVMTI extensions - API reference

Reference information for the IBM SDK extensions to the JVMTI.

Use the information in this section to control JVM functions using the IBM JVMTI interface.

Querying JVM dump options:

You can query the JVM dump options that are set for a JVM using the QueryVmDump() API.

The QueryVmDump() API has the JVMTI Extension Function identifier com.ibm.QueryVmDump. The identifier is declared as macro COM_IBM_QUERY_VM_DUMP in `ibmjvmti.h`.

To query the current JVM dump options, use:

```
jvmtiError QueryVmDump(jvmtiEnv* jvmti_env, jint buffer_size, void* options_buffer,
jint* data_size_ptr)
```

This extension returns a set of dump option specifications as ASCII strings. The syntax of the option string is the same as the **-Xdump** command-line option, with the initial **-Xdump:** omitted. See "Using the -Xdump option" on page 221. The option strings are separated by newline characters. If the memory buffer is too small to contain the current JVM dump option strings, you can expect the following results:

- The error message `JVMTI_ERROR_ILLEGAL_ARGUMENT` is returned.
- The variable for `data_size_ptr` is set to the required buffer size.

Parameters:

jvmti_env: A pointer to the JVMTI environment.

buffer_size: The size of the supplied memory buffer in bytes.

options_buffer: A pointer to the supplied memory buffer.

data_size_ptr: A pointer to a variable, used to return the total size of the option strings.

Returns:

`JVMTI_ERROR_NONE`: Success

`JVMTI_ERROR_NULL_POINTER`: The **options_buffer** or **data_size_ptr** parameters are null.

`JVMTI_ERROR_OUT_OF_MEMORY`: There is insufficient system memory to process the request.

`JVMTI_ERROR_INVALID_ENVIRONMENT`: The **jvmti_env** parameter is invalid.

`JVMTI_ERROR_WRONG_PHASE`: The extension has been called outside the JVMTI live phase.

`JVMTI_ERROR_NOT_AVAILABLE`: The dump configuration is locked because a dump is in progress.

`JVMTI_ERROR_ILLEGAL_ARGUMENT`: The supplied memory buffer in **options_buffer** is too small.

Setting JVM dump options:

You can set dump options using the same syntax as the **-Xdump** command-line option.

The SetVmDump() API has the JVMTI Extension Function identifier com.ibm.SetVmDump. The identifier is declared as macro COM_IBM_SET_VM_DUMP in `ibmjvmti.h`.

To set a JVM dump option use:

```
jvmtiError SetVmDump(jvmtiEnv* jvmti_env, char* option)
```

The dump option is passed in as an ASCII character string. Use the same syntax as the **-Xdump** command-line option, with the initial **-Xdump:** omitted. See "Using the -Xdump option" on page 221.

When dumps are in progress, the dump configuration is locked, and calls to SetVmDump() fail with a return value of JVMTI_ERROR_NOT_AVAILABLE.

Parameters:

jvmti_env: A pointer to the JVMTI environment.

option: The JVM dump option string.

Returns:

JVMTI_ERROR_NONE: Success.

JVMTI_ERROR_NULL_POINTER: The parameter **option** is null.

JVMTI_ERROR_OUT_OF_MEMORY: There is insufficient system memory to process the request.

JVMTI_ERROR_INVALID_ENVIRONMENT: The **jvmti_env** parameter is invalid.

JVMTI_ERROR_WRONG_PHASE: The extension has been called outside the JVMTI live phase.

JVMTI_ERROR_NOT_AVAILABLE: The dump configuration is locked because a dump is in progress.

JVMTI_ERROR_ILLEGAL_ARGUMENT: The parameter **option** contains an invalid **-Xdump** string.

Note: On z/OS, you might need to convert the option string from EBCDIC to ASCII before using this JVMTI extension function.

Triggering a JVM dump:

You can specify the type of dump you want using the TriggerVmDump() API.

The TriggerVmDump() API has the JVMTI Extension Function identifier com.ibm.TriggerVmDump. The identifier is declared as macro COM_IBM_TRIGGER_VM_DUMP in ibmjvmti.h.

To trigger a JVM dump, use:

```
jvmtiError TriggerVmDump(jvmtiEnv* jvmti_env, char* option)
```

Choose the type of dump required by specifying an ASCII string that contains one of the supported dump agent types. See "Dump agents" on page 224. JVMTI events are provided at the start and end of the dump.

Parameters:

jvmti_env: A pointer to the JVMTI environment.

option: A pointer to the dump type string, which can be one of the following types:

- stack
- java
- system

- `console`
- `tool`
- `heap`
- `snap`
- `ceedump` (z/OS only)

Returns:

`JVMTI_ERROR_NONE`: Success.

`JVMTI_ERROR_NULL_POINTER`: The **`option`** parameter is null.

`JVMTI_ERROR_OUT_OF_MEMORY`: There is insufficient system memory to process the request.

`JVMTI_ERROR_INVALID_ENVIRONMENT`: The **`jvmti_env`** parameter is invalid.

`JVMTI_ERROR_WRONG_PHASE`: The extension has been called outside the JVMTI live phase.

`JVMTI_ERROR_NOT_AVAILABLE`: The dump configuration is locked because a dump is in progress.

Note: On z/OS, you might need to convert the option string from EBCDIC to ASCII before using this JVMTI extension function.

Resetting JVM dump options:

Dump options can be reset using the ResetVmDump() API.

The ResetVmDump() API has the JVMTI Extension Function identifier com.ibm.ResetVmDump. The identifier is declared as macro COM_IBM_RESET_VM_DUMP in `ibmjvmti.h`.

To reset the JVM dump options to the values at JVM initialization, use:

```
jvmtiError ResetVmDump(jvmtiEnv* jvmti_env)
```

Parameters:

`jvmti_env`: The JVMTI environment pointer.

Returns:

`JVMTI_ERROR_NONE`: Success.

`JVMTI_ERROR_OUT_OF_MEMORY`: There is insufficient system memory to process the request.

`JVMTI_ERROR_INVALID_ENVIRONMENT`: The **`jvmti_env`** parameter is invalid.

`JVMTI_ERROR_WRONG_PHASE`: The extension has been called outside the JVMTI live phase.

`JVMTI_ERROR_NOT_AVAILABLE`: The dump configuration is locked because a dump is in progress.

Event function for dump start:

When a dump starts, a JVMTI event function is called.

The following JVMTI event function is called when a JVM dump starts.

```
void JNICALL VMDumpStart(jvmtiEnv *jvmti_env, JNIEnv* jni_env, char* label, char*
event, char* detail)
```

The event function provides the dump file name and the name of the JVM event that triggered the dump. For more information about dump events, see "Dump events" on page 228.

Parameters:

jvmti_env: JVMTI environment pointer.

jni_env: JNI environment pointer for the thread on which the event occurred.

label: The dump file name, including directory path.

event: The extension event name, such as com.ibm.VmDumpStart.

detail: The dump event name.

Returns:

None

Event function for dump end:

When a dump ends, a JVMTI event function is called.

The following JVMTI event function is called when a JVM dump ends:

```
void JNICALL VMDumpEnd(jvmtiEnv *jvmti_env, JNIEnv* jni_env, char* label,
char* event, char* detail)
```

This event function provides the dump file name and the name of the JVM event that triggered the dump. For more information about dump events, see "Dump events" on page 228.

Parameters:

jvmti_env: JVMTI environment pointer.

jni_env: JNI environment pointer for the thread on which the event occurred.

label: The dump file name, including directory path.

event: The extension event name, such as com.ibm.VmDumpStart.

detail: The dump event name.

Returns:

None

Setting JVM trace options:

You can set trace options for the JVM using the same syntax as the **-Xtrace** command-line option.

The SetVmTrace() API has the JVMTI Extension Function identifier com.ibm.SetVmTrace. The identifier is declared as macro COM_IBM_SET_VM_TRACE in `ibmjvmti.h`.

To set a JVM trace option, use:

```
jvmtiError SetVmTrace(jvmtiEnv* jvmti_env, char* option)
```

The trace option is passed in as an ASCII character string. Use the same syntax as the **-Xtrace** command-line option, with the initial **-Xtrace:** omitted. See "Detailed descriptions of trace options" on page 294.

Parameters:

jvmti_env: JVMTI environment pointer.

option: Enter the JVM trace option string.

Returns:

JVMTI_ERROR_NONE: Success.

JVMTI_ERROR_NULL_POINTER: The **option** parameter is null.

JVMTI_ERROR_OUT_OF_MEMORY: There is insufficient system memory to process the request.

JVMTI_ERROR_INVALID_ENVIRONMENT: The **jvmti_env** parameter is invalid.

JVMTI_ERROR_WRONG_PHASE: The extension has been called outside the JVMTI live phase.

JVMTI_ERROR_ILLEGAL_ARGUMENT: The **option** parameter contains an invalid **-Xtrace** string.

Note: On z/OS, you might need to convert the option string from EBCDIC to ASCII before using this JVMTI extension function.

Querying JRE native memory categories:

You can query the total native memory consumption of the JRE 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(). JRE native memory use is grouped under high-level memory categories, as described in the Javadump section "Native memory (NATIVEMEMINFO)" on page 246. 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 for a JVM 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 "JVM command-line options" on page 428. 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 "JVM command-line options" on page 428.

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 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 the directory is not specified on the command line, the default location for the platform is used.
- Setting the value of **useCommandLineValues** to `false` and using the **cacheDir** parameter. To accept the default location for the platform, 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 allowed 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 allowed 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;
```

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.

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 Diagnostic Tool Framework for Java

The Diagnostic Tool Framework for Java (DTFJ) is a Java application programming interface (API) from IBM used to support the building of Java diagnostic tools. DTFJ works with data from a system dump or a Javadump.

Note: The IBM Monitoring and Diagnostic Tools for Java use the DTFJ interface, and should provide all the functionality that you need. However, you can use DTFJ to write your own diagnostic tools if required.

For analysis of core 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 provided in the SDK to collect these files, see "Using jextract" on page 276. You must run **jextract** with the same SDK level, on the system that produced the system dump. The **jextract** utility compresses the dump, executable files, and libraries into a single compressed file for use in subsequent problem diagnosis.

For Java 7 SDKs on Windows and z/OS platforms, you do not need to run the **jextract** utility. For Java 6 and Java 5.0 SDKs containing versions of the IBM J9 virtual machine before V2.6, you must still run the **jextract** utility for all platforms.

To work with a Javadump, no additional processing is required.

The DTFJ API helps diagnostic tools access the following information:

- Memory locations stored in the dump (System dumps only)
- Relationships between memory locations and Java internals (System dumps only)
- Java threads running in the JVM
- Native threads held in the dump (System dumps only)
- Java classes and their class loaders that were present
- Java objects that were present in the heap (System dumps only)
- Java monitors and the objects and threads they are associated with
- Details of the workstation on which the dump was produced (System dumps only)
- Details of the Java version that was being used
- The command line that launched the JVM

If your DTFJ application requests information that is not available in the Javadump, the API will return null or throw a DataUnavailable exception. You might need to adapt DTFJ applications written to process system dumps to make them work with Javadumps.

DTFJ is implemented in pure Java and tools written using DTFJ can be cross-platform. Therefore, you can analyze a dump taken from one workstation on another (remote and more convenient) machine. For example, a dump produced on an AIX PPC workstation can be analyzed on a Windows Thinkpad.

This chapter describes DTFJ in:

API documentation for the DTFJ interface can be found here: API documentation

Using the DTFJ interface

To create applications that use DTFJ, you must use the DTFJ interface. Implementations of this interface have been written that work with system dumps from IBM SDK for Java versions 1.4.2 and later, and Javadumps from IBM SDK for Java 6 and later.

All DTFJ implementations support the same interface, but the DTFJ implementations supplied in Version 5.0 and later are different to the implementation supplied in Version 1.4.2. The DTFJ implementations have different factory class names that you must use. The DTFJ implementation supplied in Version 1.4.2 does not work with system dumps from Version 5 or later, and the DTFJ implementations supplied in Version 5 and later do not work with system dumps from Version 1.4.2.

Figure 1 on page 409 illustrates the DTFJ interface. The starting point for working with a dump is to obtain an Image instance by using the ImageFactory class supplied with the concrete implementation of the API.

Working with a system dump

The following example shows how to work with a system dump.

```
import java.io.File;
import java.util.Iterator;
import java.io.IOException;

import com.ibm.dtfj.image.CorruptData;
import com.ibm.dtfj.image.Image;
import com.ibm.dtfj.image.ImageFactory;

public class DTFJEX1 {
    public static void main(String[] args) {
        Image image = null;
        if (args.length > 0) {
            File f = new File(args[0]);
            try {
                Class factoryClass = Class
                        .forName("com.ibm.dtfj.image.j9.ImageFactory");
                ImageFactory factory = (ImageFactory) factoryClass
                        .newInstance();
                image = factory.getImage(f);
            } catch (ClassNotFoundException e) {
                System.err.println("Could not find DTFJ factory class");
                e.printStackTrace(System.err);
            } catch (IllegalAccessException e) {
                System.err.println("IllegalAccessException for DTFJ factory class");
                e.printStackTrace(System.err);
            } catch (InstantiationException e) {
                System.err.println("Could not instantiate DTFJ factory class");
                e.printStackTrace(System.err);
            } catch (IOException e) {
                System.err.println("Could not find/use required file(s)");
                e.printStackTrace(System.err);
            }
        } else {
            System.err.println("No filename specified");
        }
        if (image == null) {
            return;
        }
```

```
                Iterator asIt = image.getAddressSpaces();
                int count = 0;
                while (asIt.hasNext()) {
                    Object tempObj = asIt.next();
                    if (tempObj instanceof CorruptData) {
                        System.err.println("Address Space object is corrupt: "
                                + (CorruptData) tempObj);
                    } else {
                        count++;
                    }
                }
                System.out.println("The number of address spaces is: " + count);
            }
        }
```

In this example, the only section of code that ties the dump to a particular implementation of DTFJ is the generation of the factory class. Change the factory to use a different implementation.

The getImage() methods in ImageFactory expect one file, the `dumpfilename.zip` file produced by **jextract** (see "Using the dump viewer" on page 273). If the getImage() methods are called with two files, they are interpreted as the dump itself and the `.xml` metadata file. If there is a problem with the file specified, an IOException is thrown by getImage() and can be caught. An appropriate message issued. If a missing file is passed to the example shown, the following output is produced:

```
Could not find/use required file(s)
java.io.FileNotFoundException: core_file.xml (The system cannot find the file specified.)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:135)
        at com.ibm.dtfj.image.j9.ImageFactory.getImage(ImageFactory.java:47)
        at com.ibm.dtfj.image.j9.ImageFactory.getImage(ImageFactory.java:35)
        at DTFJEX1.main(DTFJEX1.java:23)
```

In the this case, the DTFJ implementation is expecting a dump file to exist. Different errors are caught if the file existed but was not recognized as a valid dump file.

Working with a Javadump

To work with a Javadump, change the factory class to com.ibm.dtfj.image.javacore.JCImageFactory and pass the Javadump file to the getImage() method.

```
import java.io.File;
import java.util.Iterator;
import java.io.IOException;

import com.ibm.dtfj.image.CorruptData;
import com.ibm.dtfj.image.Image;
import com.ibm.dtfj.image.ImageFactory;

public class DTFJEX2 {
    public static void main(String[] args) {
        Image image=null;

        if (args.length > 0) {
            File javacoreFile = new File(args[0]);

            try {
```

```
    Class factoryClass = Class.forName("com.ibm.dtfj.image.javacore.JCImageFactory");
    ImageFactory factory = (ImageFactory) factoryClass.newInstance();
    image = factory.getImage(javacoreFile);
} catch .....
```

The rest of the example remains the same.

After you have obtained an Image instance, you can begin analyzing the dump. The Image instance is the second instance in the class hierarchy for DTFJ illustrated by the following diagram:

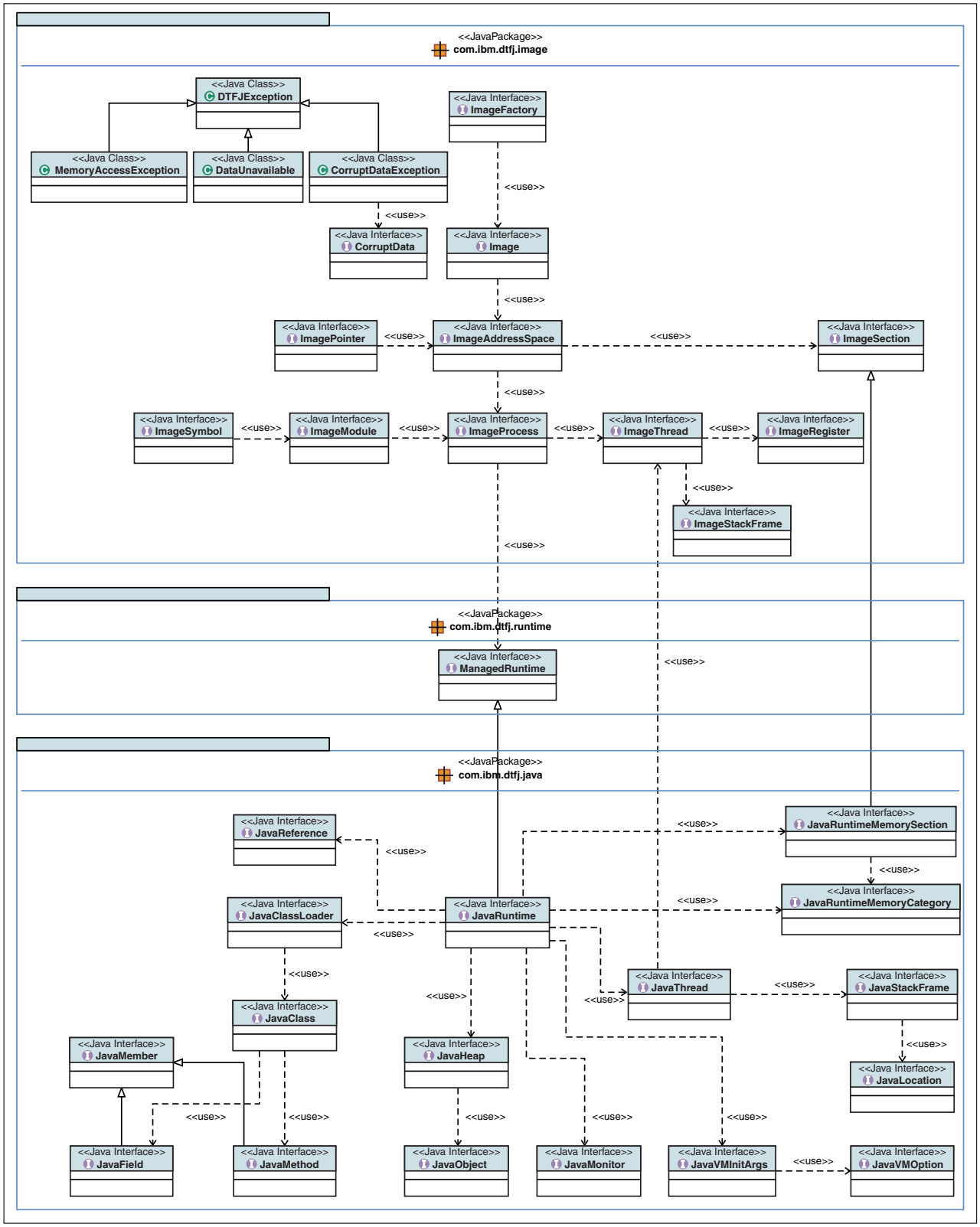


Figure 1. DTFJ interface diagram

The hierarchy displays some major points of DTFJ. First, there is a separation between the Image (the dump, a sequence of bytes with different contents on different platforms) and the Java internal knowledge.

Some things to note from the diagram:

- The DTFJ interface is separated into two parts: Classes with names that start with *Image* and classes with names that start with *Java*.
- *Image* and *Java* classes are linked using a ManagedRuntime (which is extended by JavaRuntime).
- An Image object contains one ImageAddressSpace object (or, on z/OS, possibly more).
- An ImageAddressSpace object contains one ImageProcess object (or, on z/OS, possibly more).
- Conceptually, you can apply the Image model to any program running with the ImageProcess. For the purposes of this document discussion is limited to the IBM JVM implementations.
- There is a link from a JavaThread object to its corresponding ImageThread object. Use this link to find out about native code associated with a Java thread, for example JNI functions that have been called from Java.
- If a JavaThread was not running Java code when the dump was taken, the JavaThread object has no JavaStackFrame objects. In these cases, use the link to the corresponding ImageThread object to find out what native code was running in that thread. This situation is typically the case with the JIT compilation thread and Garbage Collection threads.
- The DTFJ interface enables 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 JRE allocates native memory, the memory is associated with a high-level memory category. For more information about native memory detailed in a javadump, see "Native memory (NATIVEMEMINFO)" on page 246.

DTFJ example application

This example is a fully working DTFJ application.

For clarity, this example does not perform full error checking when constructing the main Image object and does not perform CorruptData handling in all of the iterators. In a production environment, you use the techniques illustrated in the example in the "Using the DTFJ interface" on page 406.

In this example, the program iterates through every available Java thread and checks whether it is equal to any of the available image threads. When they are found to be equal, the program displays the following message: `Found a match`.

The example demonstrates:

- How to iterate down through the class hierarchy.
- How to handle CorruptData objects from the iterators.
- The use of the .equals method for testing equality between objects.

```
import java.io.File;
import java.util.Iterator;
import com.ibm.dtfj.image.CorruptData;
import com.ibm.dtfj.image.CorruptDataException;
import com.ibm.dtfj.image.DataUnavailable;
import com.ibm.dtfj.image.Image;
import com.ibm.dtfj.image.ImageAddressSpace;
import com.ibm.dtfj.image.ImageFactory;
import com.ibm.dtfj.image.ImageProcess;
import com.ibm.dtfj.java.JavaRuntime;
import com.ibm.dtfj.java.JavaThread;
import com.ibm.dtfj.image.ImageThread;
```

```
public class DTFJEX2
{
   public static void main( String[] args )
   {
      Image image = null;
      if ( args.length > 0 )
      {
         File f = new File( args[0] );
         try
         {
            Class factoryClass = Class
                  .forName( "com.ibm.dtfj.image.j9.ImageFactory" );
            ImageFactory factory = (ImageFactory) factoryClass.newInstance( );
            image = factory.getImage( f );
         }
         catch ( Exception ex )
         { /*
             * Should use the error handling as shown in DTFJEX1.
             */
            System.err.println( "Error in DTFJEX2" );
            ex.printStackTrace( System.err );
         }
      }
      else
      {
         System.err.println( "No filename specified" );
      }

      if ( null == image )
      {
         return;
      }

      MatchingThreads( image );
   }

   public static void MatchingThreads( Image image )
   {
      ImageThread imgThread = null;

      Iterator asIt = image.getAddressSpaces( );
      while ( asIt.hasNext( ) )
      {
         System.out.println( "Found ImageAddressSpace..." );

         ImageAddressSpace as = (ImageAddressSpace) asIt.next( );

         Iterator prIt = as.getProcesses( );

         while ( prIt.hasNext( ) )
         {
            System.out.println( "Found ImageProcess..." );

            ImageProcess process = (ImageProcess) prIt.next( );

            Iterator runTimesIt = process.getRuntimes( );
            while ( runTimesIt.hasNext( ) )
            {
               System.out.println( "Found Runtime..." );
               JavaRuntime javaRT = (JavaRuntime) runTimesIt.next( );

               Iterator javaThreadIt = javaRT.getThreads( );

               while ( javaThreadIt.hasNext( ) )
               {
                  Object tempObj = javaThreadIt.next( );
```

```
                    /*
                     * Should use CorruptData handling for all iterators
                     */
                    if ( tempObj instanceof CorruptData )
                    {
                       System.out.println( "We have some corrupt data" );
                    }
                    else
                    {
                       JavaThread javaThread = (JavaThread) tempObj;
                       System.out.println( "Found JavaThread..." );
                       try
                       {
                          imgThread = (ImageThread) javaThread.getImageThread( );

                          // Now we have a Java thread we can iterator
                          // through the image threads
                          Iterator imgThreadIt = process.getThreads( );

                          while ( imgThreadIt.hasNext( ) )
                          {
                             ImageThread imgThread2 = (ImageThread) imgThreadIt
                                   .next( );
                             if ( imgThread.equals( imgThread2 ) )
                             {
                                System.out.println( "Found a match:" );
                                System.out.println( "\tjavaThread "
                                      + javaThread.getName( )
                                      + " is the same as " + imgThread2.getID( ) );
                             }
                          }
                       }
                       catch ( CorruptDataException e )
                       {
                          System.err.println( "ImageThread was corrupt: "
                                + e.getMessage( ) );
                       }
                       catch ( DataUnavailable e )
                       {
                          System.out.println( "DataUnavailable: "
                                + e.getMessage( ) );
                       }
                    }
                 }
              }
           }
        }
     }
}
```

Many DTFJ applications will follow similar models.

Using JConsole

JConsole (Java Monitoring and Management Console) is a graphical tool which allows the user to monitor and manage the behavior of Java applications.

Note: The preferred tools for monitoring Java applications are the IBM Monitoring and Diagnostic Tools for Java.

The tool is built on the `java.lang.management` API which was introduced in Java 5.0. JConsole connects to applications running on the same workstation as itself, or on a remote workstation. The applications must be configured to allow access. JConsole is not part of the core SDK, and it is experimental and unsupported.

When JConsole connects to a Java application, it reports information about the application. The details include memory usage, the running threads, and the loaded classes. This data allows you to monitor the behavior of your application and the JVM. The information is useful in understanding performance problems, memory usage issues, hangs, or deadlocks.

Setting up JConsole to monitor a Java application

1. The Java application you want to monitor must be started with command-line options which make it accessible to JConsole. The simplest set of options for monitoring are:

   ```
   -Dcom.sun.management.jmxremote.port=<port number>
   -Dcom.sun.management.jmxremote.authenticate=false
   -Dcom.sun.management.jmxremote.ssl=false
   ```

   <port number> is a free port on your workstation. In this example, the `authenticate` and `ssl` options prevent password authentication and encryption using Secure Sockets Layer (SSL). Using these options allow JConsole, or any other JMX agent, to connect to your Java application if it has access to the specified port. Only use these non-secure options in a development or testing environment. For more information about configuring security options, see http://download.oracle.com/javase/7/docs/technotes/guides/jmx/overview/connectors.html.
2. Start JConsole by typing **`jconsole`** at a command prompt. Your path must contain the `bin` directory of the SDK.
3. The JConsole **New Connection** dialog opens: Enter the host name and port number that you specified in step 1. If you are running JConsole on the same workstation as your Java application, leave the host name value as `localhost`. For a remote system, set the host field value to the host name or IP address of the workstation. Leave the **Username** and **Password** fields blank if you used the options specified in step 1.
4. Click **connect**. JConsole starts and displays the summary tab.

Setting up JConsole to monitor itself

JConsole can monitor itself. This ability is useful for simple troubleshooting of the Java environment.

1. Start JConsole by typing **`jconsole`** at a command prompt. Your path must contain the `bin` directory of the SDK.
2. The JConsole **New Connection** dialog opens: Enter `localhost:0` in the **Remote Process** field.
3. Click **connect**. JConsole starts and displays the summary tab.

Using JConsole to monitor a Java application

The JConsole summary tab shows key details of the JVM you have connected to. From here, you can select any of the other tabs for more details on a particular aspect. The Memory tab shows a history of usage of each memory pool in the JVM, – the most useful being the heap memory usage.

You can also request that a GC is carried out by clicking the **Perform GC** button. You must be connected with security options disabled as described previously, or be authenticated as a control user.

The Threads tab shows the number of threads currently running and a list of their IDs.

Clicking a thread ID shows the thread state and its current stack trace.

The Classes tab displays the current number of loaded classes and the number of classes loaded and unloaded since the application was started. Selecting the **verbose output** check box allows verbose class loading output to be switched on and off to see a list of classes that are loaded in the client JVM. The output is displayed on the `stderr` output of the client JVM.

The MBeans tab allows you to inspect the state of the platform MBeans, which provides more detail about the JVM. For more details, see "MBeans and MXBeans"

Finally, the VM tab gives information about the environment in which your Java application is running including any JVM arguments and the current class path.

Troubleshooting JConsole

JConsole is a Swing application. You might find that running JConsole on the same workstation as the Java application you want to monitor affects the performance of your Java application. You can use JConsole to connect to a JVM running on a remote workstation to reduce the affect of running JConsole on the application performance.

Because JConsole is a Java application, you can pass it Java command-line options through the application that starts JConsole by prefixing them with **–J**. For example, to change the maximum heap size that JConsole uses, add the command-line option **-J-Xmx**<*size*>.

Known Limitations

Using the local process list
: The local process list does not work. Use `localhost:<port>` in the Remote Process field to connect to a local JVM.

CPU usage in the Overview tab
: The CPU usage display does not work.

Further information

More details about JConsole and the definitions of the values it displays can be found at http://download.oracle.com/javase/7/docs/technotes/guides/management/index.html.

MBeans and MXBeans

MBeans and MXBeans can be used to provide information about the state of the Java virtual machine (JVM). IBM provides additional MXBeans that extend the monitoring and management capabilities.

MXBeans are a generalized variant of MBeans. Because MXBeans are constructed by using only a pre-defined set of data types, MXBeans can be referenced and used more easily by applications such as JConsole.

Start JConsole by running the command `jconsole` from a command line. When you connect to a running JVM, you see an MBeans tab. This tab displays a navigation tree that contains the MBeans exported by the JVM. The list of available

MBeans depends on the version of Java that you are using. The `java.lang.management` package includes MBean categories such as **Memory**, **OperatingSystem**, and **GarbageCollector**.

Clicking an MBean category in the navigation tree shows you all the related MBeans that are available. Clicking an individual MBean shows you the information that the MBean extracts from the JVM, separated into the following sections:

Attributes
: Information about the current state. You can use some MBeans to change the JVM options. For example, in the **Memory** MBean, you might select the `Verbose` option to enable VerboseGC logging output.

Operations
: Detailed information from the JVM. For example, in the **Threading** MBean, you see thread information that helps you to monitor deadlocked threads.

Notifications
: Notifications that are supported by the MBean. Applications such as JConsole receive information from the MBean by subscribing to these notifications.

Info Details about the available notifications.

IBM MXBeans

IBM provides further MXBeans to extend the monitoring and management capabilities:

GarbageCollectorMXBean
: For monitoring garbage collection operations. You can obtain data about GC collection times, heap memory usage, number of compactions, and the amount of total freed memory.

MemoryMXBean
: For monitoring memory usage, including data about maximum and minimum heap sizes, and shared class caches sizes.

MemoryPoolMXBean
: For monitoring the usage of the memory pool, where supported.

OperatingSystemMXBean
: For monitoring operating system settings such as physical and virtual memory size, processor capacity, and processor utilization.

For more information about the standard platform MBeans, see the Oracle API documentation for the `java.lang.management` package at http://download.oracle.com/javase/7/docs/api/java/lang/management/package-summary.html.

For information about IBM MXBeans, see the IBM API documentation: API documentation

Chapter 10. Reference

This part of the *Information Center* contains reference information.

The appendixes are:

Command-line options

You can specify the options on the command line while you are starting Java. They override any relevant environment variables. For example, using **-cp <dir1>** with the Java command completely overrides setting the environment variable **CLASSPATH=<dir2>**.

This chapter provides the following information:

Specifying command-line options

Although the command line is the traditional way to specify command-line options, you can also pass options to the JVM by using options files and environment variables.

The sequence of the Java options on the command line defines which options take precedence during startup. Rightmost options have precedence over leftmost options. In the following example, the **-Xjit** option takes precedence:

```
java -Xint -Xjit myClass
```

Use single or double quotation marks for command-line options only when explicitly directed to do so. Single and double quotation marks have different meanings on different platforms, operating systems, and shells. Do not use **'-X<option>'** or **"-X<option>"**. Instead, you must use **-X<option>**. For example, do not use `'-Xmx500m'` and `"-Xmx500m"`. Write this option as `-Xmx500m`.

At startup, the list of JVM arguments is constructed in the following order, with the lowest precedence first:

1. Environment variables that are described in are translated into command-line options. For example, the following environment variable adds the parameter **-Xrs** to the list of arguments:

   ```
   export IBM_NOSIGHANDLER=<non_null_string>
   ```

2. The **IBM_JAVA_OPTIONS** environment variable. You can set command-line options using this environment variable. The options that you specify with this environment variable are added to the command line when a JVM starts in that environment.

   The environment variable can contain multiple blank-delimited argument strings, but must not contain comments. For example:

   ```
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump"
   ```

   Note: The environment variable **JAVA_TOOLS_OPTIONS** is equivalent to **IBM_JAVA_OPTIONS** and is available for compatibility with JVMTI.
3. Certain options are created automatically by the JVM. These specify arguments such as search paths and version information.
4. Options that are specified on the command line. For example:

   ```
   java -Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump MyJavaClass
   ```

   The Java launcher adds some automatically generated arguments to this list, such as the names of the main class.

You can also use the **-Xoptionsfile** parameter to specify JVM options. This parameter can be used on the command line, or as part of the **IBM_JAVA_OPTIONS** environment variable. The contents of an option file are expanded in place during startup. For more information about the structure and contents of this type of file, see "-Xoptionsfile" on page 438.

To troubleshoot startup problems, you can check which options are used by a JVM. Append the following command-line option, and inspect the Javadump file that is generated:

```
-Xdump:java:events=vmstart
```

Here is an extract from a Javadump file that shows the options that are used:

```
....
    2CIUSERARG              -Xdump:java:file=/home/test_javacore.txt,events=vmstop
    2CIUSERARG              -Dtest.cmdlineOption=1
    2CIUSERARG              -XXallowvmshutdown:true
    2CIUSERARG              -Xoptionsfile=test1.test_options_file
....
```

General command-line options

Use these options to print help on assert-related options, set the search path for application classes and resources, print a usage method, identify memory leaks inside the JVM, print the product version and continue, enable verbose output, and print the product version.

-cp, -classpath <directories and compressed or .jar files separated by : (; on Windows)>
Sets the search path for application classes and resources. If **-classpath** and **-cp** are not used, and the **CLASSPATH** environment variable is not set, the user classpath is, by default, the current directory (**.**).

-help, -?
Prints a usage message.

-fullversion
Prints the build and version information for the JVM.

-showversion
Prints product version and continues.

-verbose:<*option*>**[,**<*option*>**...]**
Enables verbose output. Separate multiple options using commas. These options are available:

class
Writes an entry to `stderr` for each class that is loaded.

dynload
Provides detailed information as each bootstrap class is loaded by the JVM:

- The class name and package
- For class files that were in a .jar file, the name and directory path of the .jar
- Details of the size of the class and the time taken to load the class

The data is written out to `stderr`. An example of the output on a Windows platform follows:

```
<Loaded java/lang/String from C:\sdk\jre\lib\vm.jar>
<Class size 17258; ROM size 21080; debug size 0>
<Read time 27368 usec; Load time 782 usec; Translate time 927 usec>
```

gc Provide verbose garbage collection information.

init
Writes information to `stderr` describing JVM initialization and termination.

jni
Writes information to `stderr` describing the JNI services called by the application and JVM.

sizes
Writes information to `stderr` describing the active memory usage settings.

stack
Writes information to `stderr` describing the Java and C stack usage for each thread.

-version
Prints the full build and version information for the JVM.

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.cacheLatestUserDefinedLoader

This system property enables caching of the Latest User Defined Class Loader (LUDCL), which can improve performance when a serialized object is de-serialized.

-Dcom.ibm.cacheLatestUserDefinedLoader[true|false]
The default value for this setting is `false`.

This property addresses only the de-serialization of objects that use default serialization. For more information about Java object serialization, see http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html .

Note: This system property is superseded by "-Dcom.ibm.enableClassCaching" on page 420 in IBM SDK for z/OS, V7 service refresh 4.

-Dcom.ibm.dbgmalloc

This option provides memory allocation diagnostic information for class library native code.

-Dcom.ibm.dbgmalloc=true
When an application is started with this option, a javadump records the amount of memory allocated by the class library components. You can use this option together with the **-Xcheck:memory** option to obtain information about class library call sites and their allocation sizes. Enabling this option has an impact on throughput performance. For sample javadump output, see "Native memory (NATIVEMEMINFO)" on page 246.

-Dcom.ibm.enableClassCaching

Setting this property to `true` enables caching of the Latest User Defined Class Loader (LUDCL).

-Dcom.ibm.enableClassCaching=[true|false]
By reducing repeated lookups, Java applications that use deserialization extensively can see a performance improvement. This system property supersedes "-Dcom.ibm.cacheLatestUserDefinedLoader" on page 419, which is available only in IBM SDK for Java 7 SR3. The default value for this property is `false`.

-Dcom.ibm.jsse2.renegotiate

If your Java application uses JSSE for secure communication, you can disable TLS renegotiation by installing APAR IZ65239.

-Dcom.ibm.jsse2.renegotiate=[ALL | NONE | ABBREVIATED]

ALL Allow both abbreviated and unabbreviated (full) renegotiation handshakes.

NONE
Allow no renegotiation handshakes. This value is the default setting.

ABBREVIATED
Allow only abbreviated renegotiation handshakes.

-Dcom.ibm.lang.management.verbose

Enables verbose information from java.lang.management operations to be written to the output channel during VM operation.

-Dcom.ibm.lang.management.verbose
There are no options for this system property.

-Dcom.ibm.IgnoreMalformedInput

Invalid UTF8 or malformed byte sequences are replaced with the standard unicode replacement character `\uFFFD`.

-Dcom.ibm.IgnoreMalformedInput=true
To retain the old behavior, where invalid UTF8 or malformed byte sequences are ignored, set this system property to *true*.

-Dcom.ibm.streams.CloseFDWithStream

Determines whether the close() method of a stream object closes a native file descriptor even if the descriptor is still in use by another stream object.

-Dcom.ibm.streams.CloseFDWithStream=[true | false]
Usually, you create a FileInputStream or FileOutputStream instance by passing a String or a File object to the stream constructor method. Each stream then has a separate file descriptor. However, you can also create a stream by using

an existing FileDescriptor instance, for example one that you obtain from a RandomAccessFile instance, or another FileInputStream or FileOutputStream instance. Multiple streams can then share the same file descriptor.

If you set this option to `false`, when you use the close() method of the stream, the associated file descriptor is also closed only if it is not in use by any other streams. If you set the option to `true`, the file descriptor is closed regardless of any other streams that might still be using it.

The default setting is `true`.

Note: Before version 7 service refresh 5, the default behavior was to close the file descriptor only when all the streams that were using it were also closed. This system property exists so that you can revert to this previous default behavior if necessary. This system property will be removed in a future release, so you should adjust your applications to use the new default behavior before you upgrade to a later release.

-Dcom.ibm.tools.attach.enable

Enable the Attach API for this application.

-Dcom.ibm.tools.attach.enable=yes
The Attach API allows your application to connect to a virtual machine. Your application can then load an agent application into the virtual machine. The agent can be used to perform tasks such as monitoring the virtual machine status.

-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 IBM SDK for z/OS, V7 service refresh 5, the use of extension functions or extension elements is not allowed when Java security is enabled. This change is introduced to enhance security. The 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*.

-Dcom.ibm.zipfile.closeinputstreams

The Java.util.zip.ZipFile class allows you to create InputStreams on files held in a compressed archive.

-Dcom.ibm.zipfile.closeinputstreams=true
Under some conditions, using ZipFile.close() to close all InputStreams that have been opened on the compressed archive might result in a 56-byte-per-InputStream native memory leak. Setting the **-Dcom.ibm.zipfile.closeinputstreams=true** forces the JVM to track and close InputStreams without the memory impact caused by retaining native-backed objects. Native-backed objects are objects that are stored in native memory, rather than the Java heap. By default, the value of this system property is not enabled.

-Dfile.encoding

Use this property to define the file encoding that is required.

-Dfile.encoding=*value*

Where *value* defines the file encoding that is required.

By default the IBM GBK converter follows Unicode 3.0 standards. To force the IBM GBK converter to follow Unicode 2.0 standards, use a value of *bestfit936*.

-Dibm.jvm.bootclasspath

The value of this property is used as an additional search path.

-Dibm.jvm.bootclasspath

The value of this property is used as an additional search path, which is inserted between any value that is defined by **-Xbootclasspath/p:** and the `bootclass` path. The `bootclass` path is either the default or the one that you defined by using the **-Xbootclasspath:** option.

-Dibm.stream.nio

This option addresses the ordering of IO and NIO converters.

-Dibm.stream.nio=[true | false]

When this option is set to `true`, the NIO converters are used instead of the IO converters. By default the IO converters are used.

-Dil8n.vs

This system property enables awareness of Unicode Ideographic Variation Sequence (IVS) to draw characters, except in peered components.

-Dil8n.vs=[true]

The behavior depends on the font specified. If the font supports IVS, and has a glyph based on the combination of a base character and a variation selector character, an accurate glyph can be picked up. If not, the base character is displayed and the variation selector character is ignored. Because this option changes the behavior of the font drawing engine, the option is disabled by default. When disabled, a variation selector is displayed as an undefined character. This option is supported only for Japanese.

-Djava.compiler

Disables the Java compiler by setting to NONE.

-Djava.compiler=[NONE | j9jit<*vm_version*>**]**

Enable JIT compilation by setting to `j9jit`<*vm_version*> (Equivalent to **–Xjit**).

-Djavax.xml.namespace.QName.useCompatibleHashCodeAlgorithm

Use this property to turn off an enhanced hashing algorithm for javax.xml.namespace.QName.hashCode().

-Djavax.xml.namespace.QName.useCompatibleHashCodeAlgorithm=1.0

From Java 7 SR2 an enhanced hashing algorithm is used for javax.xml.namespace.QName.hashCode(). This algorithm can change the iteration order of items returned from hash maps. For compatibility, you can restore the earlier hashing algorithm by setting the system property **-Djavax.xml.namespace.QName.useCompatibleHashCodeAlgorithm=1.0**.

-Djdk.map.althashing.threshold

This system property controls the use of an enhanced hashing algorithm for hashed maps.

-Djdk.map.althashing.threshold=*value*

This alternative hashing algorithm is used for string keys when a hashed data structure has a capacity larger than *value*.

A value of *1* ensures that this algorithm is always used, regardless of the hashed map capacity. A value of *-1* prevents the use of this algorithm, which is the default value.

The hashed map structures affected by this threshold are: java.util.HashMap, java.util.Hashtable, java.util.LinkedHashMap, java.util.WeakHashMap, and java.util.concurrent.ConcurrentHashMap.

The capacity of a hashed map is related to the number of entries in the map, multiplied by the load factor. Because the capacity of a hashed map is rounded up to the next power of two, setting the threshold to intermediate values has no affect on behavior. For example, threshold values of 600, 700, and 1000 have the same effect. However, values of 1023 and 1024 cause a difference in behavior. For a more detailed description of the capacity and load factor, see http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html.

When entries are removed from a hashed map the capacity does not shrink. Therefore, if the map ever exceeds the threshold to use alternative hashing for Strings, the map always uses alternative hashing for Strings. This behavior does not change, even if entries are later removed or the map is emptied using clear().

The enhanced hashing algorithm is available from Java 7 SR2

-Djdk.reflect.allowGetCallerClass

Use this option to re-enable the sun.reflect.Reflection.getCallerClass(int depth) method.

-Djdk.reflect.allowGetCallerClass

To enhance security, the sun.reflect.Reflection.getCallerClass(int depth) method is not supported from Version 7 service refresh 6. Use the sun.reflect.Reflection.getCallerClass() method instead. This method always uses a depth of 2.

If you use the sun.reflect.Reflection.getCallerClass(int depth) method in your application, an UnsupportedOperationException exception is thrown.

Note: You can use this option to re-enable support for the sun.reflect.Reflection.getCallerClass(int depth) method, but this option will be removed in a future release. You must set the option on the command line when you start the application; you cannot set it from within the application at run time.

You can use this option in several ways. The following methods enable the option:

- **-Djdk.reflect.allowGetCallerClass**
- **-Djdk.reflect.allowGetCallerClass=**
- **-Djdk.reflect.allowGetCallerClass=true** (**true** is not case sensitive, so **TRUE** or **tRuE** are equally valid)

The option is disabled by default, but you can also specifically disable it by using one of the following methods:

- **-Djdk.reflect.allowGetCallerClass=false**
- **-Djdk.reflect.allowGetCallerClass=***any_other_value*

-Djdk.xml.entityExpansionLimit

This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the number of entity expansions in an XML document.

-Djdk.xml.entityExpansionLimit=*value*

Where *value* is a positive integer. The default value is 64,000.

A value of *0* or a negative number sets no limits.

You can also set this limit by adding the following line to your `jaxp.properties` file:

```
jdk.xml.entityExpansionLimit=<value>
```

Related reference:

"-Djdk.xml.maxGeneralEntitySizeLimit"
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the maximum size of a general entity.

"-Djdk.xml.maxOccur" on page 425
This option provides limits for Java API for XML (JAXP) processing. This option defines the maximum number of content model nodes that can be created in a grammar.

"-Djdk.xml.maxParameterEntitySizeLimit" on page 425
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the maximum size of a parameter entity.

"-Djdk.xml.totalEntitySizeLimit" on page 426
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the total size of all entities that include general and parameter entities.

-Djdk.xml.maxGeneralEntitySizeLimit

This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the maximum size of a general entity.

To protect an application from malformed XML, set this value to the minimum size possible.

-Djdk.xml.maxGeneralEntitySizeLimit=*value*

Where *value* is the maximum size that is allowed for a general entity. The default value is 0.

A value of *0* or a negative number sets no limits.

You can also set this limit by adding the following line to your `jaxp.properties` file:

```
jdk.xml.maxGeneralEntitySizeLimit=<value>
```

Related reference:

"-Djdk.xml.entityExpansionLimit" on page 424
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the number of entity expansions in an XML document.

"-Djdk.xml.maxOccur"
This option provides limits for Java API for XML (JAXP) processing. This option defines the maximum number of content model nodes that can be created in a grammar.

"-Djdk.xml.maxParameterEntitySizeLimit"
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the maximum size of a parameter entity.

"-Djdk.xml.totalEntitySizeLimit" on page 426
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the total size of all entities that include general and parameter entities.

-Djdk.xml.maxOccur

This option provides limits for Java API for XML (JAXP) processing. This option defines the maximum number of content model nodes that can be created in a grammar.

When building a grammar for a W3C XML schema, use this option to limit the number of content model nodes that can be created when the schema defines attributes that can occur multiple times.

-Djdk.xml.maxOccur=*value*

Where *value* is a positive integer. The default value is 5,000.

A value of *0* or a negative number sets no limits.

You can also set this limit by adding the following line to your `jaxp.properties` file:

```
jdk.xml.maxoccur=<value>
```

Related reference:

"-Djdk.xml.entityExpansionLimit" on page 424
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the number of entity expansions in an XML document.

"-Djdk.xml.maxGeneralEntitySizeLimit" on page 424
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the maximum size of a general entity.

"-Djdk.xml.maxParameterEntitySizeLimit"
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the maximum size of a parameter entity.

"-Djdk.xml.totalEntitySizeLimit" on page 426
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the total size of all entities that include general and parameter entities.

-Djdk.xml.maxParameterEntitySizeLimit

This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the maximum size of a parameter entity.

To protect an application from malformed XML, set this value to the minimum size possible.

-Djdk.xml.maxParameterEntitySizeLimit=*value*

Where *value* is the maximum size that is allowed for a parameter entity. The default value is 0.

A value of *0* or a negative number sets no limits.

You can also set this limit by adding the following line to your `jaxp.properties` file:

```
jdk.xml.maxParameterEntitySizeLimit=<value>
```

Related reference:

"-Djdk.xml.entityExpansionLimit" on page 424
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the number of entity expansions in an XML document.

"-Djdk.xml.maxGeneralEntitySizeLimit" on page 424
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the maximum size of a general entity.

"-Djdk.xml.maxOccur" on page 425
This option provides limits for Java API for XML (JAXP) processing. This option defines the maximum number of content model nodes that can be created in a grammar.

"-Djdk.xml.totalEntitySizeLimit"
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the total size of all entities that include general and parameter entities.

-Djdk.xml.totalEntitySizeLimit

This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the total size of all entities that include general and parameter entities.

-Djdk.xml.totalEntitySizeLimit=*value*

Where *value* is the collective size of all entities. The default value is 5x10^7.

A value of *0* or a negative number sets no limits.

You can also set this limit by adding the following line to your `jaxp.properties` file:

```
jdk.xml.totalEntitySizeLimit=<value>
```

Related reference:

"-Djdk.xml.entityExpansionLimit" on page 424
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the number of entity expansions in an XML document.

"-Djdk.xml.maxGeneralEntitySizeLimit" on page 424
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the maximum size of a general entity.

"-Djdk.xml.maxOccur" on page 425
This option provides limits for Java API for XML (JAXP) processing. This option defines the maximum number of content model nodes that can be created in a grammar.

"-Djdk.xml.maxParameterEntitySizeLimit" on page 425
This option provides limits for Java API for XML (JAXP) processing. Use this option to limit the maximum size of a parameter entity.

-Dsun.awt.keepWorkingSetOnMinimize

The **-Dsun.awt.keepWorkingSetOnMinimize=true** system property stops the JVM trimming an application when it is minimized.

-Dsun.awt.keepWorkingSetOnMinimize=true
When a Java application using the Abstract Windowing Toolkit (AWT) is minimized, the default behavior is to "trim" the "working set". The working set is the application memory stored in RAM. Trimming means that the working set is marked as being available for swapping out if the memory is required by another application. The advantage of trimming is that memory is available for other applications. The disadvantage is that a "trimmed" application might experience a delay as the working set memory is brought back into RAM.

The default behavior is to trim an application when it is minimized.

-Dsun.net.client.defaultConnectTimeout

Specifies the default value for the connect timeout for the protocol handlers used by the java.net.URLConnection class.

-Dsun.net.client.defaultConnectTimeout=<*value in milliseconds*>
The default value set by the protocol handlers is -1, which means that no timeout is set.

When a connection is made by an applet to a server and the server does not respond properly, the applet might seem to hang. The delay might also cause the browser to hang. The apparent hang occurs because there is no network connection timeout. To avoid this problem, the Java Plug-in has added a default value to the network timeout of 2 minutes for all HTTP connections. You can override the default by setting this property.

-Dsun.net.client.defaultReadTimeout

Specifies the default value for the read timeout for the protocol handlers used by the java.net.URLConnection class when reading from an input stream when a connection is established to a resource.

-Dsun.net.client.defaultReadTimeout=<*value in milliseconds*>
The default value set by the protocol handlers is -1, which means that no timeout is set.

-Dsun.nio.MaxDirectMemorySize

Limits the native memory size for nio Direct Byte Buffer objects to the value specified.

-Dsun.nio.MaxDirectMemorySize=<*value*>
Specify <*value*> in bytes.

-Dsun.rmi.transport.tcp.connectionPool

Enables thread pooling for the RMI ConnectionHandlers in the TCP transport layer implementation.

-Dsun.rmi.transport.tcp.connectionPool=*val*
val is either *true* or a value that is not null.

-Dswing.useSystemFontSettings

This option addresses compatibility problems for Swing programs.

-Dswing.useSystemFontSettings=[false]
By default, Swing programs running with the Windows Look and Feel render with the system font set by the user instead of a Java-defined font. As a result, fonts differ from the fonts in earlier releases. This option addresses compatibility problems like these for programs that depend on the old behavior. By setting this option, v1.4.1 fonts and those of earlier releases are the same for Swing programs running with the Windows Look and Feel.

JVM command-line options

Use these options to configure your JVM. The options prefixed with **-X** are nonstandard.

Options that relate to the JIT are listed under "JIT and AOT command-line options" on page 448. Options that relate to the Garbage Collector are listed under "Garbage Collector command-line options" on page 453.

-X

Displays help on nonstandard options.

-X Displays help on nonstandard options.

-Xaggressive

Enables performance optimizations.

-Xaggressive
Enables performance optimizations and new platform exploitation that are expected to be the default in future releases.

-Xargencoding

Include Unicode escape sequences in the argument list.

-Xargencoding
You can put Unicode escape sequences in the argument list. This option is set to off by default.

-Xbootclasspath

Sets the search path for bootstrap classes and resources.

-Xbootclasspath:<*directories and compressed or Java archive files separated by : (; on* **Windows***)*>
The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xbootclasspath/a:

Appends to the end of the search path for bootstrap classes.

-Xbootclasspath/a:<*directories and compressed or Java archive files separated by : (; on* **Windows**)>
Appends the specified directories, compressed files, or .jar files to the end of the bootstrap class path. The default is to search for bootstrap classes and resources in the internal VM directories and `.jar` files.

-Xbootclasspath/p:

Adds a prefix to the search path for bootstrap classes.

-Xbootclasspath/p:<*directories and compressed or Java archive files separated by : (; on* **Windows**)>
Adds a prefix of the specified directories, compressed files, or Java archive files to the front of the bootstrap class path. Do not deploy applications that use the **-Xbootclasspath:** or the **-Xbootclasspath/p:** option to override a class in the standard API. The reason is that such a deployment contravenes the Java 2 Runtime Environment binary code license. The default is to search for bootstrap classes and resources in the internal VM directories and `.jar` files.

-Xcheck

You can use the **-Xcheck** option to run checks during JVM startup, such as memory checks or checks on JNI functions.

-Xcheck:<*option*>
The options available are detailed in separate topics.

-Xcheck:classpath:

Displays a warning message if an error is discovered in the class path.

-Xcheck:classpath
Checks the classpath and reports if an error is discovered; for example, a missing directory or JAR file.

-Xcheck:gc:

Runs additional checks on garbage collection.

-Xcheck:gc[:<*scan options*>][:<*verify options*>][:<*misc options*>]
By default, no checks are made. See the output of **-Xcheck:gc:help** for more information.

-Xcheck:jni:

Runs additional checks for JNI functions.

-Xcheck:jni[**:help**][:<*option*>=<*value*>]
This option is equivalent to **-Xrunjnichk**. By default, no checks are made.

-Xcheck:memory:

Identifies memory leaks inside the JVM.

-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 that are 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 if user JNI code or the JVM is responsible for memory corruption. Corruption in the JVM **subAllocator** heap suggests that the JVM is causing the problem; corruption in the user-allocated memory suggests that user code is corrupting memory. Typically, user and JVM allocated memory are interleaved.

zero
Newly allocated blocks are set to 0 instead of being filled with the `0xE7E7xxxxxxxxE7E7` pattern. Setting these blocks to 0 helps you to determine whether a callsite is expecting zeroed memory, in which case the allocation request is followed by `memset(pointer, 0, size)`.

Note: The **-Xcheck:memory** option cannot be used in the **-Xoptionsfile**.

-Xcheck:vm:

Performs additional checks on the JVM.

-Xcheck:vm[:<*option*>]
By default, no checking is performed. For more information, run **-Xcheck:vm:**`help`.

-Xclassgc

Enables dynamic unloading of classes by the JVM. Garbage collection of class objects occurs only on class loader changes.

-Xclassgc
Dynamic unloading is the default behavior. To disable dynamic class unloading, use the **-Xnoclassgc** option.

-Xcompressedrefs

Enables the use of compressed references.

-Xcompressedrefs
(64-bit only) To disable compressed references, use the **-Xnocompressedreferences** option. For more information, see "Compressed references" on page 27.

From service refresh 4, compressed references are enabled by default for all platforms other than z/OS, but only when the value of the **-Xmx** garbage collector option is less than or equal to 25 GB. Before service refresh 4, compressed references were disabled by default for all platforms.

You cannot include this option in an options file. You must specify this option on the command line, or by using the **IBM_JAVA_OPTIONS** environment variable.

-XCEEHDLR

Controls 31-bit z/OS JVM Language Environment condition handling.

-XCEEHDLR (31-bit z/OS only)

This option is used to control 31-bit z/OS JVM Language Environment condition handling. 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 JRE continues to register POSIX signal handlers for the following signals:

- SIGABRT
- SIGINT
- SIGQUIT
- SIGTERM

Signal chaining using the `libjsig.so` library is not supported.

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.

-Xdiagnosticscollector

Enables the Diagnostics Collector.

-Xdiagnosticscollector[:settings=<filename>]
See "The Diagnostics Collector" on page 328 for more information. The settings option allows you to specify a different Diagnostics Collector settings file to use instead of the default `dc.properties` file in the JRE.

-Xdisablejavadump

Turns off Javadump generation on errors and signals.

-Xdisablejavadump
By default, Javadump generation is enabled.

-Xdump

Use the **-Xdump** option to add and remove dump agents for various JVM events, update default dump settings (such as the dump name), and limit the number of dumps that are produced.

-Xdump
See "Using dump agents" on page 221 for more information.

-Xenableexplicitgc

This options tells the VM to trigger a garbage collection when a call is made to System.gc().

-Xenableexplicitgc
Signals to the VM that calls to System.gc() trigger a garbage collection. This option is enabled by default.

-Xfastresolve

Tune performance by improving the resolution time for classes.

-Xfastresolve<*n*>
This option is used to tune performance by improving the resolution time for classes when the field count exceeds the threshold specified by <*n*>. If profiling tools show significant costs in field resolution, change the threshold until the costs are reduced. If you enable this option, additional memory is used when the threshold is exceeded.

-Xfuture

Turns on strict class-file format checks.

-Xfuture
Use this flag when you are developing new code because stricter checks will become the default in future releases. By default, strict format checks are disabled.

-Xifa

Enables Java applications to run on IFAs if they are available.

-Xifa:<*on* | *off* | *force*> **(z/OS only)**
z/OS V1R6 or later can run Java applications on a new type of special-purpose assist processor called the *System z Application Assist Processor* (zAAP). The zAAP is also known as an IFA (Integrated Facility for Applications).

The **-Xifa** option enables Java applications to run on IFAs if they are available. The default value for the **-Xifa** option is *on*. Only Java code and system native methods can be on IFA processors.

The *force* option is obsolete and should not be used. This option is superseded by the **SYS1.PARMLIB(IEAOPTxx) PROJECTCPU=YES** parameter, which is available on all supported levels of z/OS. **Xifa:***force* can be used for testing purposes when a zAAP is not available, but can have a negative performance impact.

-Xiss

Sets the initial stack size for Java threads.

-Xiss<*size*>
By default, the stack size is set to 2 KB. Use the **-verbose:sizes** option to output the value that the VM is using.

-Xjarversion

Produces output information about the version of each .jar file.

-Xjarversion
Produces output information about the version of each .jar file in the class path, the boot class path, and the extensions directory. Version information is taken from the Implementation-Version and Build-Level properties in the manifest of the .jar file.

Note: The **-Xjarversion** option cannot be used in the **-Xoptionsfile**.

-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 8096 bytes.

-Xlinenumbers

Displays line numbers in stack traces for debugging.

-Xlinenumbers
See also **-Xnolinenumbers**. By default, line numbers are on.

-Xlockword

Test whether performance optimizations are negatively impacting an application.

-Xlockword:<options>

-Xlockword:[mode=all|mode=default]
See "Testing JVM optimizations" on page 192.

-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.

-Xlog

Enables message logging.

-Xlog[:help]|[:<option>]

Optional parameters are:

- **help** - details the options available
- **error** - turns on logging for all JVM 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 JVM information messages
- **warn** - turns on logging for all JVM warning messages
- **config** - turns on logging for all JVM configuration messages
- **all** - turns on logging for all JVM messages
- **none** - turns off logging for all JVM messages

Note: Changes made to message logging using the **-Xlog** option do not affect messages written to the standard error stream (stderr).
The options **all**, **none** and **help** must be used on their own and cannot be combined. However, the other options can be grouped. For example, to include error, vital and warning messages use **-Xlog:error,vital,warn**. For message details see "JVM messages" on page 465.

-Xlp

Requests the JVM to allocate the Java object heap with large pages.

-Xlp[<*size*>**]**

z/OS: Requests the JVM to allocate the Java object heap 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 using **-Xlp[<size>]** is only supported on the 64-bit SDK for z/OS, not the 31-bit JVM for z/OS.

1M pageable pages, when available, are the default size for the object heap and the code cache. The options that control these sizes are **Xlp:codecache:** and **-Xlp:objectheap:**.

On z/OS, **-Xlp**<*size*> only supports a large page size of 2G and 1M (nonpageable).

For more information, see "Configuring large page memory allocation" on page 154.

AIX, Linux, Windows, and z/OS: If a <*size*> is specified, the JVM attempts to allocate the JIT code cache memory using pages of that size. If unsuccessful, or if executable pages of that size are not supported, 1M pageable is attempted. If 1M pageable is not available, the JIT code cache memory is allocated using the default or smallest available executable page size.

-Xlp1M uses a 1M pageable size for the code cache, when available. **-Xlp2G** sets the object heap size, but generates a warning that 2G nonpageable pages cannot be used for the code cache. Use the **-Xlp:objectheap=2G,nonpageable** option to avoid the warning.

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.

There is no error message issued when the operating system does not have sufficient resources to satisfy the request. This limitation and a workaround for

verifying the page size used can be found in Known limitations.
For more information, see “Configuring large page memory allocation” on page 154.

-Xlp:codecache:

Requests the JVM to allocate the JIT code cache by using large page sizes.

-Xlp:codecache:pagesize=<size>,pageable (z/OS)

If the requested large page size is not available, the JVM starts, but the JIT code cache is allocated using a platform-defined size. A warning is displayed when the requested page size is not available.

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 more information, see “Configuring large page memory allocation” on page 154.

1M pageable pages, when available, are the default size for the code cache.

z/OS: The **-Xlp:codecache:pagesize=<size>,pageable** option supports only a large page size of 1M pageable large pages. The use of 1M pageable large pages for the JIT code cache can improve the runtime performance of some Java applications. A page size of 4K can also be used.

-Xlp:objectheap:

Requests the JVM to allocate the Java object heap by using large page sizes.

-Xlp:objectheap:pagesize=<*size*>**,[non]pageable (z/OS)**

If the requested large page size is not available, the JVM starts, but the Java object heap is allocated using a platform-defined size. A warning is displayed when the requested page size is not available.

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, see “Configuring large page memory allocation” on page 154.

z/OS: The `[non]pageable` argument defines the type of memory to allocate for the Java object heap.

1M pageable pages, when available, are the default size for the object heap.

Supported large page sizes are 2G nonpageable, 1M nonpageable, and 1M pageable.

A page size of 4K can also be used.

All platforms: There is no error message issued when the operating system does not have sufficient resources to satisfy the request. This limitation and a workaround for verifying which page size is used can be found in “Known issues and limitations” on page 476.

-Xmso

Sets the initial stack size for operating system threads.

-Xmso<*size*>

The default value can be determined by running the command:

```
java -verbose:sizes
```

The maximum value for the stack size varies according to platform and specific machine configuration. If you exceed the maximum value, a `java/lang/OutOfMemoryError` message is reported.

-Xnoagent

Disables support for the old JDB debugger.

-Xnoagent

Disables support for the old JDB debugger.

-Xnoclassgc

Disables class garbage collection.

-Xnoclassgc

This option switches off garbage collection of storage associated with Java classes that are no longer being used by the JVM. The default behavior is as defined by **-Xclassgc**. Enabling this option is not recommended except under the direction of the IBM Java support team. The reason is the option can cause unlimited native memory growth, leading to out-of-memory errors.

-Xnocompressedrefs

Disables the use of compressed references.

-Xnocompressedrefs

(64-bit only) To enable compressed references, use the **-Xcompressedreferences** option. For more information, see "Compressed references" on page 27.

From service refresh 4, compressed references are enabled by default for all platforms other than z/OS, but only when the value of the **-Xmx** garbage collector option is less than or equal to 25 GB. Before service refresh 4, compressed references were disabled by default for all platforms.

You cannot include this option in an options file. You must specify this option on the command line, or by using the **IBM_JAVA_OPTIONS** environment variable.

-Xnolinenumbers

Disables the line numbers for debugging.

-Xnolinenumbers

See also **-Xlinenumbers**. By default, line number are on.

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**.

-Xnosigcatch

Disables JVM signal handling code.

-Xnosigcatch

See also **-Xsigcatch**. By default, signal handling is enabled.

-Xnosigchain

Disables signal handler chaining.

-Xnosigchain
See also **-Xsigchain**. By default, the signal handler chaining is enabled, except for z/OS.

-Xoptionsfile

Specifies a file that contains JVM options and definitions.

-Xoptionsfile=<*file*>
where <*file*> contains options that are processed as if they had been entered directly as command-line options. By default, a user option file is not used.

Here is an example of an options file:

```
#My options file
-X<option1>
-X<option2>=\
<value1>,\
<value2>
-D<sysprop1>=<value1>
```

The options file does not support these options:

- **-assert**
- **-fullversion**
- **-help**
- **-showversion**
- **-version**
- **-Xcompressedrefs**
- **-Xcheck:memory**
- **-Xjarversion**
- **-Xoptionsfile**

Although you cannot use **-Xoptionsfile** recursively within an options file, you can use **-Xoptionsfile** multiple times on the same command line to load more than one options files.

Some options use quoted strings as parameters. Do not split quoted strings over multiple lines using the line continuation character '\'. The '¥' character is not supported as a line continuation character. For example, the following example is not valid in an options file:

```
-Xevents=vmstop,exec="cmd /c \
echo %pid has finished."
```

The following example is valid in an options file:

```
-Xevents=vmstop, \
exec="cmd /c echo %pid has finished."
```

-Xoss

Sets the maximum Java stack size for any thread.

-Xoss<*size*>
Recognized but deprecated. Use **-Xss** and **-Xmso** instead. The maximum value for the stack size varies according to platform and specific machine configuration. If you exceed the maximum value, a `java/lang/OutOfMemoryError` message is reported.

-Xrdbginfo

Loads the remote debug information server with the specified host and port.

-Xrdbginfo:<*host*>**:**<*port*>

By default, the remote debug information server is disabled.

-Xrs

Disables signal handling in the JVM.

-Xrs

Setting **-Xrs** prevents the Java run time environment from handling any internally or externally generated signals such as SIGSEGV and SIGABRT. Any signals raised are handled by the default operating system handlers. Disabling signal handling in the JVM reduces performance by approximately 2-4%, depending on the application.

-Xrs:sync

As with **-Xrs**, the use of **-Xrs:sync** reduces performance by approximately 2-4%, depending on the application.

-Xscdmx

Use the **-Xscdmx** option to control the size of the class debug area when creating a shared class cache.

-Xscdmx<*size*>

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.

size is expressed as an absolute value.

A class debug area is still created if you use the **-Xnolinenumbers** option with the **-Xscdmx** option on the command line.

-Xscmx

Specifies cache size.

-Xscmx<*size*>

This option applies only if a cache is being created and no cache of the same name exists. The default cache size is platform-dependent. You can find out the size value being used by adding **-verbose:sizes** as a command-line argument. Minimum cache size is 4 KB. Maximum cache size is platform-dependent. The size of cache that you can specify is limited by the amount of physical memory and paging space available to the system. The virtual address space of a process is shared between the shared classes cache and the Java heap. Increasing the maximum size of the Java heap reduces the size of the shared classes cache that you can create.

-Xshareclasses

Enables class sharing.

-Xshareclasses:<*suboptions*>

This option can take a number of suboptions, some of which are cache utilities. Cache utilities perform the required operation on the specified cache, without starting the VM. You can combine multiple suboptions, separated by commas, but the cache utilities are mutually exclusive.

You can use the following suboptions with the **-Xshareclasses** option:

cacheDir=<*directory*>

Sets the directory in which cache data is read and written. By default, <*directory*> is `/tmp/javasharedresources` on Linux, AIX, z/OS, and IBM i.

You must have sufficient permissions in <*directory*>. Nonpersistent caches are stored in shared memory and have control files that describe the location of the memory. Control files are stored in a `javasharedresources` subdirectory of the **cacheDir** specified. Do not move or delete control files in this directory. The **listAllCaches** utility, the **destroyAll** utility, and the **expire** suboption work only in the scope of a given **cacheDir**.

cacheDirPerm=<*permission*>
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.

cacheRetransformed
Enables caching of classes that are transformed by using the JVMTI RetransformClasses function. See "JVMTI redefinition and retransformation of classes" on page 354 for more information.

destroy (Utility option)
Destroys a cache that is specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. A cache can be destroyed only if all JVMs using it have shut down and the user has sufficient permissions.

destroyAll (Utility option)
Tries to destroy all caches available using the specified **cacheDir** and **nonpersistent** suboptions. A cache can be destroyed only if all JVMs using it have shut down and the user has sufficient permissions.

disableBCI
Turns off BCI support. This option can be used to override **-XX:ShareClassesEnableBCI**. For more information, see "JVM -XX command-line options" on page 446.

enableBCI

Allows a JVMTI ClassFileLoadHook event to be triggered every time, for classes that are loaded from the cache. This mode also prevents caching of classes that are modified by JVMTI agents. For more information about this option, see "Using the JVMTI ClassFileLoadHook with cached classes" on page 353. 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 that is 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 that is 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.

expire=<*time in minutes*> **(Utility option)**
Destroys all caches that are unused for the time that is specified before loading shared classes. This option is not a utility option because it does not cause the JVM to exit.

groupAccess
Sets operating system permissions on a new cache to allow group access to the cache. Group access can be set only when permitted by the operating system **umask** setting. The default is user access only.

help
Lists all the command-line options.

listAllCaches (Utility option)
Lists all the compatible and incompatible caches that exist in the specified cache directory. If you do not specify **cacheDir**, the default directory is used. Summary information, such as Java version and current usage, is displayed for each cache.

Note: Some features, when enabled, result in the creation of caches that cannot be shared with caches that are created when the feature is disabled. The multitenancy support is one such example. In this situation, you can have more than one cache with the same name. The output from the **listAllCaches** option has a `feature` column which lists the feature that created the cache, usually `default`. For multitenancy support, the feature is `mt`, and the cache is listed in the `Incompatible shared caches` section of the output.

mprotect=[all | default | none]
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 small performance cost. Specifying `all` ensures that all the cache pages are protected, including the header. Specifying `none` disables the page protection.

modified=<*modified context*>
Used when a JVMTI agent is installed that might modify bytecode at run time. If you do not specify this suboption and a bytecode modification agent is installed, classes are safely shared with an extra performance cost. The <*modified context*> is a descriptor chosen by the user; for example, *myModification1*. This option partitions the cache, so that only JVMs using context *myModification1* can share the same classes. For instance, if you run an application with a modification context and then run it again with a different modification context, all classes are stored twice in the cache. See "Dealing with runtime bytecode modification" on page 351 for more information.

name=<*name*>
Connects to a cache of a given name, creating the cache if it does not exist. This option is also used to indicate the cache that is to be modified by

cache utilities; for example, **destroy**. Use the **listAllCaches** utility to show which named caches are currently available. If you do not specify a name, the default name “sharedcc_%u” is used. "%u" in the cache name inserts the current user name. You can specify “%g” in the cache name to insert the current group name.

Note: Some features, when enabled, result in the creation of caches that cannot be shared with caches that were created when the feature was disabled. The multitenancy support is one such example. In this situation, you can have more than one cache with the same name.

noaot
Disables caching and loading of AOT code.

noBootclasspath
Disables the storage of classes loaded by the bootstrap class loader in the shared classes cache. Often used with the SharedClassURLFilter API to control exactly which classes are cached. See “Using the SharedClassHelper API” on page 359 for more information about shared class filtering.

none
Added to the end of a command line, disables class data sharing. This suboption overrides class sharing arguments found earlier on the command line.

nonfatal
Allows the JVM to start even if class data sharing fails. Normal behavior for the JVM is to refuse to start if class data sharing fails. If you select **nonfatal** and the shared classes cache fails to initialize, the JVM attempts to connect to the cache in read-only mode. If this attempt fails, the JVM starts without class data sharing.

printAllStats (Utility option)
Displays detailed information about the contents of the cache that is specified in the **name=**<*name*> suboption. If the name is not specified, statistics are displayed about the default cache. Every class is listed in chronological order with a reference to the location from which it was loaded. See “printAllStats utility” on page 366 for more information.

printStats[=<*data_types*>**] (Utility option)**
Displays summary information for the cache that is specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. The most useful information that is displayed is how full the cache is and how many classes it contains. Stale classes are classes that are updated on the file system and which the cache has therefore marked as "stale". Stale classes are not purged from the cache and can be reused.

Specify one or more data types, which are separated by a plus symbol (+), to additionally see more detailed information about the cache content. Data types include AOT data, class paths, and ROMMethods.See “printStats utility” on page 362 for more information.

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.

readonly
Opens an existing cache with read-only permissions. The JVM does not create a new cache with this suboption. Opening a cache read-only prevents the JVM from making any updates to the cache. It also allows the JVM to connect to caches created by other users or groups without requiring write access. By default, this suboption is not specified.

reset
Causes a cache to be destroyed and then re-created when the JVM starts up. This option can be added to the end of a command line as **-Xshareclasses:reset**.

silent
Disables all shared class messages, including error messages. Unrecoverable error messages, which prevent the JVM from initializing, are displayed.

verbose
Gives detailed output on the cache I/O activity, listing information about classes that are stored and found. Each class loader is given a unique ID (the bootstrap loader is always 0) and the output shows the class loader hierarchy at work, where class loaders must ask their parents for a class before they can load it themselves. It is typical to see many failed requests; this behavior is expected for the class loader hierarchy. The standard option **-verbose:class** also enables class sharing verbose output if class sharing is enabled.

verboseAOT
Enables verbose output when compiled AOT code is being found or stored in the cache. AOT code is generated heuristically. You might not see any AOT code that is generated at all for a small application. You can disable AOT caching using the **noaot** suboption. See the IBM JVM Messages Guide for a list of the messages produced.

verboseHelper
Enables verbose output for the Java Helper API. This output shows you how the Helper API is used by your class loader.

verboseIO
Gives detailed output on the cache I/O activity, listing information about classes that are stored and found. Each class loader is given a unique ID (the bootstrap loader is always 0) and the output shows the class loader hierarchy at work, where class loaders must ask their parents for a class before they can load it themselves. It is typical to see many failed requests; this behavior is expected for the class loader hierarchy.

-Xsigcatch
Enables VM signal handling code.

-Xsigcatch
See also **-Xnosigcatch**. By default, signal handling is enabled.

-Xsigchain

Enables signal handler chaining.

-Xsigchain
See also **-Xnosigchain**. By default, signal handler chaining is enabled.

-Xsignal:posixSignalHandler=cooperativeShutdown

This option affects the behavior of JVM signal handlers.

-Xsignal:posixSignalHandler=cooperativeShutdown
When the JVM signal handlers for SIGSEGV, SIGILL, SIGBUS, SIGFPE, SIGTRAP, and SIGABRT end a process, they call exit(), by default. In this case, the z/OS Language Environment is not aware that the JVM ended abnormally.

With **-Xsignal:posixSignalHandler=cooperativeShutdown**, the JVM no longer uses exit() to end the process from the signal handlers. Instead, the JVM behaves in one of the following ways:

- In response to a z/OS hardware exception, the JVM uses return().
- In response to signals raised or injected by software, the JVM ends the enclave with abend 3565.

Language Environment detects that the JVM is ending abnormally and initiates Resource Recovery Services. For more information about signal handlers, see "Signals used by the JVM" on page 124.

-Xsignal:userConditionHandler=percolate (31-bit z/OS only)

This option results in similar behavior to 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.\

-Xsignal:userConditionHandler=percolate
As with the **-XCEEHDLR** option, 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**

-Xss

Sets the maximum stack size for Java threads.

-Xss<*size*>
The default is 256 KB for 32-bit JVMs and 512 KB for 64-bit JVMs. The maximum value varies according to platform and specific machine configuration. If you exceed the maximum value, a `java/lang/OutOfMemoryError` message is reported.

-Xssi

Sets the stack size increment for Java threads.

-Xssi<*size*>
When the stack for a Java thread becomes full it is increased in size by this value until the maximum size (**-Xss**) is reached. The default is 16 KB.

-Xthr

-Xthr:<*suboptions*>

-Xthr:<AdaptSpin|noAdaptSpin>
This tuning option is available to test whether performance optimizations are negatively impacting an application. See "Testing JVM optimizations" on page 192.

-Xthr:minimizeUserCPU
Minimizes user-mode CPU usage in thread synchronization where possible. The reduction in CPU usage might be a trade-off in exchange for decreased performance.

-Xthr:<secondarySpinForObjectMonitors|noSecondarySpinForObjectMonitors>
This tuning option is available to test whether performance optimizations are negatively impacting an application. See "Testing JVM optimizations" on page 192.

-Xtrace

Trace options.

-Xtrace[**:help**] | [:<*option*>=<*value*>, ...]
See "Controlling the trace" on page 293 for more information.

-Xtune:elastic

This option turns on JVM function that accommodates changes in the machine configuration dynamically at run time.

-Xtune:elastic
Such changes might include the number of processors, or the amount of installed RAM.

-Xtune:virtualized

Optimizes JVM function for virtualized environments, such as a cloud.

-Xtune:virtualized
Optimizes JVM function for virtualized environments, such as a cloud.

-Xverify

Use this option to enable or disable the verifier.

-Xverify[:<*option*>**]**
With no parameters, enables the verifier, which is the default. Therefore, if used on its own with no parameters, for example, **-Xverify**, this option does nothing. Optional parameters are as follows:

- **all** - enable maximum verification
- **none** - disable the verifier
- **remote** - enables strict class-loading checks on remotely loaded classes

The verifier is on by default and must be enabled for all production servers. Running with the verifier off is not a supported configuration. If you encounter problems and the verifier was turned off using **-Xverify:none**, remove this option and try to reproduce the problem.

-Xzero

Enables reduction of the memory footprint of the Java runtime environment when concurrently running multiple Java invocations.

-Xzero[:<*option*>**]**

-Xzero might not be appropriate for all types of applications because it changes the implementation of `java.util.ZipFile`, which might cause extra memory usage. **-Xzero** includes the optional parameters:

- **j9zip** - enables the j9zip sub option
- **noj9zip** - disables the j9zip sub option
- **sharezip** - enables the sharezip sub option
- **nosharezip** - disables the sharezip sub option
- **sharebootzip** - enables the sharebootzip sub option
- **nosharebootzip** - disables the sharebootzip 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 suboption is not a requirement for **sharezip**; however, if **j9zip** is not enabled, only the bootstrap .jar files 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** suboption, note that this suboption allows every opened .zip file and .jar file to store the j9zip cache in shared memory, so you might fill the shared memory when opening multiple new .zip files and .jar files. The affected API is `java.util.zip.ZipFile` (superclass of `java.util.jar.JarFile`). The .zip and .jar files do not have to be on a class path.
- **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.

.

JVM -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.

-XXallowvmshutdown

This option is provided as a workaround for customer applications that cannot shut down cleanly, as described in APAR IZ59734.

-XXallowvmshutdown:[false|true]
Customers who need this workaround should use **-XXallowvmshutdown:false**. The default option is **-XXallowvmshutdown:true**.

-XX:codecachetotal

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.

-XX:codecachetotal=<*size*>
This option is an alias for the "-Xcodecachetotal" on page 449 option.

-XX:MaxDirectMemorySize

Sets the maximum size for an nio direct buffer.

-XX:MaxDirectMemorySize=<*size*>
When you set a value for this property, the size cannot exceed that value. If you do not set a value, a soft limit of 64 MB is set. The JVM automatically expands this soft limit in 32 MB chunks, as required.

-XXnosuballoc32bitmem

When compressed references are used with a 64-bit JVM on z/OS, this option forces the JVM to use 31-bit memory allocation functions provided by z/OS.

-XXnosuballoc32bitmem

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 this option is not specified, the JVM uses an allocation strategy for 31-bit memory that reserves a region of 31-bit virtual memory.

-XX:ShareClassesEnableBCI

This option is equivalent to **-Xshareclasses:enableBCI**.

-XX:ShareClassesEnableBCI

-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 "JVM command-line options" on page 428.

-XX:-StackTraceInThrowable

This option removes stack traces from exceptions.

-XX:-StackTraceInThrowable
By default, stack traces are available in exceptions. Including a stack trace in exceptions requires walking the stack and that can affect performance. Removing stack traces from exceptions can improve performance but can also make problems harder to debug.

When this option is enabled, Throwable.getStackTrace() returns an empty array and the stack trace is displayed when an uncaught exception occurs. Thread.getStackTrace() and Thread.getAllStackTraces() are not affected by this option.

-XX:[+|-]UseCompressedOops (64-bit only)

This option enables or disables compressed references in 64-bit JVMs, and is provided to help when porting applications from the Oracle JVM to the IBM JVM. This option might not be supported in subsequent releases.

-XX:[+|-]UseCompressedOops

The **-XX:+UseCompressedOops** option enables compressed references in 64-bit JVMs. The **-XX:+UseCompressedOops** option is similar to specifying **-Xcompressedrefs**, which is detailed in the topic "JVM command-line options" on page 428.

The **-XX:-UseCompressedOops** option prevents the use of compressed references in 64-bit JVMs.

-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 class loader is not synchronized. For information about parallel capable class loaders, see java.lang.ClassLoader.registerAsParallelCapable(). Note that this option might cause a deadlock if class loaders 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.

JIT and AOT command-line options

Use these JIT and AOT compiler command-line options to control code compilation.

For options that take a <*size*> parameter, suffix the number with "k" or "K" to indicate kilobytes, "m" or "M" to indicate megabytes, or "g" or "G" to indicate gigabytes.

For more information about JIT and AOT, see JIT and AOT problem determination"JIT and AOT problem determination" on page 322.

-Xaot

Use this option to control the behavior of the AOT compiler.

-Xaot[:<*parameter*>=<*value*>**, ...]**

With no parameters, enables the AOT compiler. The AOT compiler is enabled by default but is not active unless shared classes are enabled. Using this option on its own has no effect. The following parameters are useful:

count=<*n*>
Where <*n*> is the number of times a method is called before it is compiled or loaded from an existing shared class cache. For example, setting `count=0` forces the AOT compiler to compile everything on first execution.

limitFile=(<*filename*>,<*m*>,<*n*>)
Compile or load only the methods listed on lines <*m*> to <*n*> in the specified limit file. Methods not listed in the limit file and methods listed on lines outside the range are not compiled or loaded.

loadExclude=<*methods*>
Do not load methods beginning with <*methods*>.

loadLimit=<*methods*>
Load methods beginning with <*methods*> only.

loadLimitFile=(<*filename*>,<*m*>,<*n*>)
Load only the methods listed on lines <*m*> to <*n*> in the specified limit file. Methods not listed in the limit file and methods listed on lines outside the range are not loaded.

verbose
Reports information about the AOT and JIT compiler configuration and method compilation.

-Xcodecache

This option is used to tune performance.

-Xcodecache<*size*>
This option sets the size of each block of memory that is allocated to store the native code of compiled Java methods. By default, this size is selected internally according to the processor architecture and the capability of your system. If profiling tools show significant costs in *trampolines*, that is a good reason to change the size until the costs are reduced. Changing the size does not mean always increasing the size. The option provides the mechanism to tune for the correct size until hot interblock calls are eliminated. A reasonable starting point to tune for the optimal size is (totalNumberByteOfCompiledMethods * 1.1).

Note: Trampolines are small pieces of code that facilitate the transition between two compiled methods that are far apart from each other in memory.

-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*>
See "JIT and AOT command-line options" on page 448 for more information about the <*size*> parameter.

By default, the total size of the JIT code cache is determined by your operating system, architecture, and the version of IBM SDK Java Technology Edition that you are using. Long-running, complex, server-type applications can fill the JIT code cache, which can cause performance problems because not all of the important methods can be JIT-compiled. Use the **-Xcodecachetotal** option to increase the maximum code cache size beyond the default setting, to a setting that suits your application.

The value that you specify is rounded up to a multiple of the code cache block size, as specified by the "-Xcodecache" on page 449 option. If you specify a value for the **-Xcodecachetotal** optoin that is smaller than the default setting, that value is ignored.

When you use this option, the maximum size limit for the JIT data cache, which holds metadata about compiled methods, is increased proportionally to support the additional JIT compilations.

The maximum size limits, for both the JIT code and data caches, that are in use by the JVM are shown in Javadump output. Look for lines that begin with `1STSEGLIMIT`. Use this information together with verbose JIT tracing to determine suitable values for this option on your system. For example Javadump output, see "Storage Management (MEMINFO)" on page 248.

Related reference:

"-Xjit" on page 451
Use the JIT compiler command line option to produce verbose JIT trace output.

Related information:

"Using Javadump" on page 240
Javadump produces files that contain diagnostic information that is related to the JVM and a Java application that is captured at a point during execution. For example, the information can be about the operating system, the application environment, threads, stacks, locks, and memory.

-Xcomp (z/OS only)

Forces methods to be compiled by the JIT compiler on their first use.

-Xcomp
The use of this option is deprecated; use **-Xjit:count=0** instead.

-XcompilationThreads

Use this option to specify the number of compilation threads that are used by the JIT compiler.

-XcompilationThreads<*number of threads*>
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 and JRE for Java v6.

```
<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
```

-Xint

This option makes the JVM use the Interpreter only, disabling the Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilers.

-Xint
By default, the JIT compiler is enabled. By default, the AOT compiler is enabled, but is not used by the JVM unless shared classes are also enabled.

-Xjit

Use this option to control the behavior of the JIT compiler.

-Xjit[:<*parameter*>**=**<*value*>**, ...]**
With no parameters, enables the JIT compiler. The JIT compiler is enabled by default, so using this option on its own has no effect. Useful parameters are:

count=<*n*>
Where <*n*> is the number of times a method is called before it is compiled. For example, setting `count=0` forces the JIT compiler to compile everything on first execution.

exclude={<*method*>**}**
Excludes the specified method from compilation.

limitFile=(<*filename*>**,** <*m*>**,** <*n*>**)**
Compile only the methods listed on lines <*m*> to <*n*> in the specified limit file. Methods not listed in the limit file and methods listed on lines outside the range are not compiled.

optlevel=[noOpt | cold | warm | hot | veryHot | scorching]
Forces the JIT compiler to compile all methods at a specific optimization level. Specifying **optlevel** might have an unexpected effect on performance, including reduced overall performance.

verbose[={compileStart|compileEnd}]
Reports information about the JIT and AOT compiler configuration and method compilation.

The **={compileStart|compileEnd}** option reports when the JIT starts to compile a method, and when it ends.

vlog=<*filename*>
Sends verbose output to a file. If you do not specify this parameter, the output is sent to the standard error output stream (stderr).

Related tasks:

"Diagnosing a JIT or AOT problem" on page 322
Occasionally, valid bytecodes might compile into invalid native code, causing the Java program to fail. By determining whether the JIT or AOT compiler is faulty and, if so, *where* it is faulty, you can provide valuable help to the Java service team.

-Xnoaot

This option turns off the AOT compiler and disables the use of AOT-compiled code.

-Xnoaot
By default, the AOT compiler is enabled but is active only when shared classes are also enabled. Using this option does not affect the JIT compiler.

-Xnojit

This option turns off the JIT compiler.

-Xnojit
By default, the JIT compiler is enabled. This option does not affect the AOT compiler.

-Xquickstart

This option causes the JIT compiler to run with a subset of optimizations.

-Xquickstart
The effect is faster compilation times that improve startup time, but longer running applications might run slower. When the AOT compiler is active (both shared classes and AOT compilation enabled), **-Xquickstart** causes all methods to be AOT compiled. The AOT compilation improves the startup time of subsequent runs, but might reduce performance for longer running applications. **-Xquickstart** can degrade performance if it is used with long-running applications that contain hot methods. The implementation of **-Xquickstart** is subject to change in future releases. By default, **-Xquickstart** is disabled..

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.

-XsamplingExpirationTime

Use this option to disable JIT sampling after a specified amount of time.

-XsamplingExpirationTime<*time*>
Disables the JIT sampling thread after <*time*> seconds. When the JIT sampling thread is disabled, no processor cycles are used by an idle JVM.

-Xscmaxaot

Optionally applies a maximum number of bytes in the class cache that can be used for AOT data.

-Xscmaxaot<*size*>
This option is useful if you want a certain amount of cache space guaranteed for non-AOT data. If this option is not specified, the maximum limit for AOT data is the amount of free space in the cache. The value of this option must not be smaller than the value of **-Xscminaot** and must not be larger than the value of **-Xscmx**.

-Xscmaxjitdata

Optionally applies a maximum number of bytes in the class cache that can be used for JIT data.

-Xscmaxjitdata<*x*>
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**.

-Xscminaot

Optionally applies a minimum number of bytes in the class cache to reserve for AOT data.

-Xscminaot<*size*>
If this option is not specified, no space is reserved for AOT data. However, AOT data is still written to the cache until the cache is full or the **-Xscmaxaot** limit is reached. The value of this option must not exceed the value of **-Xscmx**

or **-Xscmaxaot**. The value of **-Xscminaot** must always be considerably less than the total cache size, because AOT data can be created only for cached classes. If the value of **-Xscminaot** equals the value of **-Xscmx**, no class data or AOT data can be stored.

-Xscminjitdata

Optionally applies a minimum number of bytes in the class cache to reserve for JIT data.

-Xscminjitdata<*x*>

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 **-Xscmaxjit** 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.

Garbage Collector command-line options

Use these Garbage Collector command-line options to control garbage collection.

You might need to read "Memory management" on page 23 to understand some of the references that are given here.

The **-verbose:gc** option detailed in "Verbose garbage collection logging" on page 334 is the main diagnostic aid that is available for runtime analysis of the Garbage Collector. However, additional command-line options are available that affect the behavior of the Garbage Collector and might aid diagnostic data collection.

For options that take a <*size*> parameter, suffix the number with "k" or "K" to indicate kilobytes, "m" or "M" to indicate megabytes, or "g" or "G" to indicate gigabytes.

For options that take a <*percentage*> parameter, use a number from 0 to 1, for example, 50% is 0.5.

-Xalwaysclassgc

Always perform dynamic class unloading checks during global collection.

-Xalwaysclassgc

The default behavior is as defined by **-Xclassgc**.

-Xclassgc

Enables dynamic unloading of classes by the JVM. Garbage collection of class objects occurs only on class loader changes.

-Xclassgc

Dynamic unloading is the default behavior. To disable dynamic class unloading, use the **-Xnoclassgc** option.

-Xcompactexplicitgc

Enables full compaction each time System.gc() is called.

-Xcompactexplicitgc

Enables full compaction each time System.gc() is called.

-Xcompactgc

Compacts on all garbage collections (system and global).

-Xcompactgc
The default (no compaction option specified) makes the GC compact based on a series of triggers that attempt to compact only when it is beneficial to the future performance of the JVM.

-Xconcurrentbackground

Specifies the number of low-priority background threads attached to assist the mutator threads in concurrent mark.

-Xconcurrentbackground<*number*>
The default is 0 on Linux System z and 1 on all other platforms.

-Xconcurrentlevel

Specifies the allocation "tax" rate.

-Xconcurrentlevel<*number*>
This option indicates the ratio between the amount of heap allocated and the amount of heap marked. The default is 8.

-Xconcurrentslack

Attempts to keep the specified amount of the heap space free in concurrent collectors by starting the concurrent operations earlier.

-Xconcurrentslack<*size*>
Using this option can sometimes alleviate pause time problems in concurrent collectors at the cost of longer concurrent cycles, affecting total throughput. The default value is 0, which is optimal for most applications.

-Xconmeter

This option determines the usage of which area, LOA (Large Object Area) or SOA (Small Object Area), is metered and hence which allocations are taxed during concurrent mark.

-Xconmeter:<*soa* | *loa* | *dynamic*>
Using **-Xconmeter:soa** (the default) applies the allocation tax to allocations from the small object area (SOA). Using **-Xconmeter:loa** applies the allocation tax to allocations from the large object area (LOA). If **-Xconmeter:dynamic** is specified, the collector dynamically determines which area to meter based on which area is exhausted first, whether it is the SOA or the LOA.

-Xdisableexcessivegc

Disables the throwing of an OutOfMemory exception if excessive time is spent in the GC.

-Xdisableexcessivegc
Disables the throwing of an OutOfMemory exception if excessive time is spent in the GC.

-Xdisableexplicitgc

Disables System.gc() calls.

-Xdisableexplicitgc

Many applications still make an excessive number of explicit calls to System.gc() to request garbage collection. In many cases, these calls degrade performance through premature garbage collection and compactions. However, you cannot always remove the calls from the application.

The **-Xdisableexplicitgc** parameter allows the JVM to ignore these garbage collection suggestions. Typically, system administrators use this parameter in applications that show some benefit from its use.

By default, calls to System.gc() trigger a garbage collection.

-Xdisablestringconstantgc

Prevents strings in the string intern table from being collected.

-Xdisablestringconstantgc
Prevents strings in the string intern table from being collected.

-Xenableexcessivegc

If excessive time is spent in the GC, the option returns null for an allocate request and thus causes an OutOfMemory exception to be thrown.

-Xenableexcessivegc

The OutOfMemory exception is thrown only when the heap has been fully expanded and the time spent is making up at least 95%. This behavior is the default.

You can control the percentage that triggers an excessive GC event with the **-Xgc:excessiveGCratio** option. For more information, see "-Xgc."

-Xenablestringconstantgc

Enables strings from the string intern table to be collected.

-Xenablestringconstantgc
This option is on by default.

-Xgc

Options that change the behavior of the Garbage Collector (GC).

-Xgc:<excessiveGCratio | minContractPercent | maxContractPercent | overrideHiresTimerCheck | verboseFormat>

excessiveGCratio=*value*
Where *value* is a percentage. The default value is 95. This option can be used only when **-Xenableexcessivegc** is set. For more information, see "-Xenableexcessivegc."

minContractPercent=<n>
The minimum percentage of the heap that can be contracted at any given time.

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%.

overrideHiresTimerCheck
When the JVM starts, the GC checks that the operating system can meet the timer resolution requirements for the requested target pause time. Typically, this check correctly identifies operating systems that can deliver adequate time resolution. However, in some cases the operating system provides a more conservative answer than strictly necessary for GC pause time management, which prevents startup. Specifying the **-Xgc:overrideHiresTimerCheck** option causes the GC to ignore the answer returned by the operating system. The JVM starts, but GC pause time management remains subject to operating system performance, which might not provide adequate timer resolution.

Note: Use this option with caution, and only when you are unable to use a supported operating system.

verboseFormat=<format>
Accepted values are:

- *default*: The default verbose garbage collection format for this release of the SDK. See "Verbose garbage collection logging" on page 334.
- *deprecated*: The verbose garbage collection format available in earlier releases of the SDK. For more information, see the Diagnostics Guide: http://www.ibm.com/developerworks/java/jdk/diagnosis/index.html.

-Xgcpolicy

Controls the behavior of the Garbage Collector.

-Xgcpolicy:< balanced | gencon | optavgpause | optthruput >

gencon
The **gencon** policy (default) uses a concurrent mark phase combined with generational garbage collection to help minimize the time that is spent in any garbage collection pause. This policy is particularly useful for applications with many short-lived objects, such as transactional applications. Pause times can be significantly shorter than with the **optthruput** policy, while still producing good throughput. Heap fragmentation is also reduced.

balanced
The **balanced** policy uses mark, sweep, compact and generational style garbage collection. The concurrent mark phase is disabled; concurrent garbage collection technology is used, but not in the way that concurrent mark is implemented for other policies. The **balanced** 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 tries to avoid global collections by matching object allocation and survival rates. If you have problems with application pause times that are caused by global garbage collections, particularly compactions, this policy might improve application performance. For more information about this policy, including when to use it, see "Balanced Garbage Collection policy" on page 39.

optavgpause
The **optavgpause** policy uses concurrent mark and concurrent sweep phases. Pause times are shorter than with **optthruput**, but application throughput is reduced because some garbage collection work is taking place while the application is running. Consider using this policy if you have a large heap size (available on 64-bit platforms), because this policy limits the effect of increasing heap size on the length of the garbage collection pause. However, if your application uses many short-lived objects, the **gencon** policy might produce better performance.

subpool
The **subpool** policy is deprecated and is now an alias for **optthruput**. Therefore, if you use this option, the effect is the same as **optthruput**.

optthruput
The **optthruput** policy disables the concurrent mark phase. The application stops during global garbage collection, so long pauses can occur. This configuration is typically used for large-heap applications when high application throughput, rather than short garbage collection pauses, is the

main performance goal. If your application cannot tolerate long garbage collection pauses, consider using another policy, such as **gencon**.

-Xgcthreads

Sets the number of threads that the Garbage Collector uses for parallel operations.

-Xgcthreads<*number*>

The 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 *n-1*, where *n* is the number of reported CPUs, up to a maximum of 64. Where SMT or hyperthreading is in place, the number of reported CPUs is larger than the number of physical CPUs. Likewise, where virtualization is in place, the number of reported CPUs is the number of virtual CPUs assigned to the operating system. 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 to more than the default setting.

On systems running multiple JVMs or in LPAR environments where multiple JVMs can share the same physical CPUs, you might want to restrict the number of GC threads used by each JVM. The restriction helps prevent the total number of parallel operation GC threads for all JVMs exceeding the number of physical CPUs present, when multiple JVMs perform garbage collection at the same time.

-Xgcworkpackets

Specifies the total number of work packets available in the global collector.

-Xgcworkpackets<*number*>

If you do not specify a value, the collector allocates a number of packets based on the maximum heap size.

-Xloa

Allocates a large object area (LOA).

-Xloa

Objects are allocated in this LOA rather than the SOA. By default, the LOA is enabled for all GC policies except for subpool, where the LOA is not available.

-Xloainitial

Specifies the initial percentage (between 0 and 0.95) of the current tenure space allocated to the large object area (LOA).

-Xloainitial<*percentage*>

The default value is 0.05, which is 5%.

-Xloamaximum

Specifies the maximum percentage (between 0 and 0.95) of the current tenure space allocated to the large object area (LOA).

-Xloamaximum<*percentage*>

The default value is 0.5, which is 50%.

-Xloaminimum

Specifies the minimum percentage (between 0 and 0.95) of the current tenure space allocated to the large object area (LOA).

-Xloaminimum<*percentage*>

The LOA does not shrink to less than this value. The default value is 0, which is 0%.

-Xmaxe

Sets the maximum amount by which the garbage collector expands the heap.

-Xmaxe<*size*>

Typically, the garbage collector expands the heap when the amount of free space falls to less than 30% (or by the amount specified using **-Xminf**), by the amount required to restore the free space to 30%. The **-Xmaxe** option limits the expansion to the specified value; for example `-Xmaxe10M` limits the expansion to 10 MB. By default, there is no maximum expansion size.

-Xmaxf

Specifies the maximum percentage of heap that must be free after a garbage collection.

-Xmaxf<*percentage*>

If the free space exceeds this amount, the JVM tries to shrink the heap. The default value is 0.6 (60%).

-Xmaxt

Specifies the maximum percentage of time to be spent in Garbage Collection.

-Xmaxt<*percentage*>

If the percentage of time exceeds this value, the JVM tries to expand the heap. The default value is 13%.

-Xmca

Sets the expansion step for the memory allocated to store the RAM portion of loaded classes.

-Xmca<*size*>

Each time more memory is required to store classes in RAM, the allocated memory is increased by this amount. By default, the expansion step is 32 KB. Use the **-verbose:sizes** option to determine the value that the VM is using. If the expansion step size you choose is too large, `OutOfMemoryError` is reported. The exact value of a "too large" expansion step size varies according to the platform and the specific machine configuration.

-Xmco

Sets the expansion step for the memory allocated to store the ROM portion of loaded classes.

-Xmco<*size*>

Each time more memory is required to store classes in ROM, the allocated memory is increased by this amount. By default, the expansion step is 128 KB. Use the **-verbose:sizes** option to determine the value that the VM is using. If the expansion step size you choose is too large, `OutOfMemoryError` is reported. The exact value of a "too large" expansion step size varies according to the platform and the specific machine configuration.

-Xmine

Sets the minimum amount by which the Garbage Collector expands the heap.

-Xmine<*size*>

Typically, the garbage collector expands the heap by the amount required to restore the free space to 30% (or the amount specified using **-Xminf**). The **-Xmine** option sets the expansion to be at least the specified value; for example, `-Xmine50M` sets the expansion size to a minimum of 50 MB. By default, the minimum expansion size is 1 MB.

-Xminf

Specifies the minimum percentage of heap to remain free after a garbage collection.

-Xminf<*percentage*>
If the free space falls to less than this amount, the JVM attempts to expand the heap. The default value is 30%.

-Xmint

Specifies the minimum percentage of time to spend in Garbage Collection.

-Xmint<*percentage*>
If the percentage of time drops to less than this value, the JVM tries to shrink the heap. The default value is 5%.

-Xmn

Sets the initial and maximum size of the new area to the specified value when using **-Xgcpolicy:gencon**.

-Xmn<*size*>
Equivalent to setting both **-Xmns** and **-Xmnx**. If you set either **-Xmns** or **-Xmnx**, you cannot set **-Xmn**. If you try to set **-Xmn** with either **-Xmns** or **-Xmnx**, the VM does not start, returning an error. By default, **-Xmn** is not set. If the scavenger is disabled, this option is ignored.

-Xmns

Sets the initial size of the new area to the specified value when using **-Xgcpolicy:gencon**.

-Xmns<*size*>
By default, this option is set to 25% of the value of the **-Xms** option. This option returns an error if you try to use it with **-Xmn**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using. If the scavenger is disabled, this option is ignored.

-Xmnx

Sets the maximum size of the new area to the specified value when using **-Xgcpolicy:gencon**.

-Xmnx<*size*>
By default, this option is set to 25% of the value of the **-Xmx** option. This option returns an error if you try to use it with **-Xmn**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using. If the scavenger is disabled, this option is ignored.

-Xmo

Sets the initial and maximum size of the old (tenured) heap to the specified value when using **-Xgcpolicy:gencon**.

-Xmo<*size*>
Equivalent to setting both **-Xmos** and **-Xmox**. If you set either **-Xmos** or **-Xmox**, you cannot set **-Xmo**. If you try to set **-Xmo** with either **-Xmos** or **-Xmox**, the VM does not start, returning an error. By default, **-Xmo** is not set.

-Xmoi

Sets the amount the Java heap is incremented when using **-Xgcpolicy:gencon**.

-Xmoi<*size*>
If set to zero, no expansion is allowed. By default, the increment size is calculated on the expansion size, set by **-Xmine** and **-Xminf**.

-Xmos

Sets the initial size of the old (tenure) heap to the specified value when using **-Xgcpolicy:gencon**.

-Xmos<*size*>
: By default, this option is set to 75% of the value of the **-Xms** option. This option returns an error if you try to use it with **-Xmo**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

-Xmox

Sets the maximum size of the old (tenure) heap to the specified value when using **-Xgcpolicy:gencon**.

-Xmox<*size*>
: By default, this option is set to the same value as the **-Xmx** option. This option returns an error if you try to use it with **-Xmo**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

-Xmr

Sets the size of the Garbage Collection "remembered set".

-Xmr<*size*>
: The Garbage Collection "remembered set" is a list of objects in the old (tenured) heap that have references to objects in the new area. By default, this option is set to 16 K.

-Xmrx

Sets the remembered maximum size setting.

-Xmrx<*size*>
: Sets the remembered maximum size setting.

-Xms

Sets the initial Java heap size.

-Xms<*size*>
: <*size*> can be specified in megabytes (m) or gigabytes (g). For example: **-Xms2g** sets an initial Java heap size of 2GB. The minimum size is 1 MB.

You can also use the **-Xmo** option.

If the scavenger is enabled, **-Xms** >= **-Xmn** + **-Xmo**.

If the scavenger is disabled, **-Xms** >= **-Xmo**.

Note: The **-Xmo** option is not supported by the balanced garbage collection policy.

-Xmx

Sets the maximum memory size for the application (**-Xmx** >= **-Xms**).

-Xmx<*size*>
: <*size*> can be specified in megabytes (m) or gigabytes (g). For example: **-Xmx2g** sets a maximum heap size of 2GB.

  For information about default values, see "Default settings for the JVM" on page 474.

If you are allocating the Java heap with large pages, read the information provided for the "-Xlp" on page 435 option.

Examples of the use of **-Xms** and **-Xmx**:

-Xms2m -Xmx64m
Heap starts at 2 MB and grows to a maximum of 64 MB.

-Xms100m -Xmx100m
Heap starts at 100 MB and never grows.

-Xms20m -Xmx1024m
Heap starts at 20 MB and grows to a maximum of 1 GB.

-Xms50m
Heap starts at 50 MB and grows to the default maximum.

-Xmx256m
Heap starts at default initial value and grows to a maximum of 256 MB.

If you exceed the limit set by the **-Xmx** option, the JVM generates an `OutofMemoryError`.

-Xnoclassgc

Disables class garbage collection.

-Xnoclassgc
This option switches off garbage collection of storage associated with Java classes that are no longer being used by the JVM. The default behavior is as defined by **-Xclassgc**. Enabling this option is not recommended except under the direction of the IBM Java support team. The reason is the option can cause unlimited native memory growth, leading to out-of-memory errors.

-Xnocompactexplicitgc

Disables compaction on System.gc() calls.

-Xnocompactexplicitgc
Compaction takes place on global garbage collections if you specify **-Xcompactgc** or if compaction triggers are met. By default, compaction is enabled on calls to System.gc().

-Xnocompactgc

Disables compaction on all garbage collections (system or global).

-Xnocompactgc
By default, compaction is enabled.

-Xnoloa

Prevents allocation of a large object area; all objects are allocated in the SOA.

-Xnoloa
See also **-Xloa**.

-Xsoftmx

This option sets a "soft" maximum limit for the initial size of the Java heap.

-Xsoftmx<*size*>
Use the **-Xmx** option to set a "hard" limit for the maximum size of the heap. By default, **-Xsoftmx** is set to the same value as **-Xmx**. The value of **-Xms** must be less than, or equal to, the value of **-Xsoftmx**. See the introduction to this topic for more information about specifying <*size*> parameters.

You can set this option on the command line, then modify it at run time by using the MemoryMXBean.setMaxHeapSize() method in the com.ibm.lang.management API. By using this API, Java applications can

dynamically monitor and adjust the heap size as required. This function can be useful in virtualized or cloud environments, for example, where the available memory might change dynamically to meet business needs. When you use the API, you must specify the value in bytes, such as `2147483648` instead of `2g`.

For example, you might set the initial heap size to 1 GB and the maximum heap size to 8 GB. You might set a smaller value, such as 2 GB, for **-Xsoftmx**, to limit the heap size that is used initially:

```
–Xms1g –Xsoftmx2g –Xmx8g
```

You can then use the com.ibm.lang.management API from within a Java application to increase the **-Xsoftmx** value during run time, as load increases. This change allows the application to use more memory than you specified initially.

If you reduce the **-Xsoftmx** value, the garbage collector 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 to less than the new limit, the heap will not grow beyond that limit.

When the heap shrinks, the garbage collector might release memory. The ability of the operating system to reclaim and use this memory varies based on the capabilities of the operating system.

Notes:

- When using **-Xgcpolicy:gencon**, **-Xsoftmx** applies only to the non-nursery portion of the heap. In some cases the heap grows to greater than the **-Xsoftmx** value because the nursery portion grows, making the heap size exceed the limit that is set. See **-Xmn** for limiting the nursery size.
- When using **-Xgcpolicy:metronome**, **-Xsoftmx** is ignored because the Metronome garbage collector does not support contraction or expansion of the heap.

-Xsoftrefthreshold

Sets the value used by the garbage collector to determine the number of garbage collections after which a soft reference is cleared if its referent has not been marked.

-Xsoftrefthreshold<*number*>

The default is 32, meaning that the soft reference is cleared after 32 * (percentage of free heap space) garbage collection cycles where its referent was not marked. For example, if **-Xsoftrefthreshold** is set to 32, and the heap is 50% free, soft references are cleared after 16 garbage collection cycles.

-Xtgc

Provides garbage collection tracing options.

-Xtgc:<*arguments*>

<*arguments*> is a comma-separated list containing one or more of the following arguments:

backtrace

Before a garbage collection, a single line is printed containing the name of the master thread for garbage collection, as well as the value of the osThread slot in the J9VMThread structure.

compaction
Prints extra information showing the relative time spent by threads in the “move” and “fixup” phases of compaction

concurrent
Prints extra information showing the activity of the concurrent mark background thread

dump
Prints a line of output for every free chunk of memory in the system, including "dark matter" (free chunks that are not on the free list for some reason, typically because they are too small). Each line contains the base address and the size in bytes of the chunk. If the chunk is followed in the heap by an object, the size and class name of the object is also printed. This argument has a similar effect to the **terse** argument.

freeList
Before a garbage collection, prints information about the free list and allocation statistics since the last garbage collection. Prints the number of items on the free list, including "deferred" entries (with the scavenger, the unused space is a deferred free list entry). For TLH and non-TLH allocations, prints the total number of allocations, the average allocation size, and the total number of bytes discarded during allocation. For non-TLH allocations, also included is the average number of entries that were searched before a sufficiently large entry was found.

parallel
Produces statistics on the activity of the parallel threads during the mark and sweep phases of a global garbage collection.

scavenger
Prints extra information after each scavenger collection. A histogram is produced showing the number of instances of each class, and their relative ages, present in the survivor space. The information is obtained by performing a linear walk-through of the space.

terse
Dumps the contents of the entire heap before and after a garbage collection. For each object or free chunk in the heap, a line of trace output is produced. Each line contains the base address, "a" if it is an allocated object, and "f" if it is a free chunk, the size of the chunk in bytes, and, if it is an object, its class name.

-Xverbosegclog

Causes **-verbose:gc** output to be written to a specified file.

-Xverbosegclog[:<*file*>**[,<X>,<Y>]]**
If the file cannot be found, **-verbose:gc** tries to create the file, and then continues as normal if it is successful. If it cannot create the file (for example, if an invalid filename is passed into the command), it redirects the output to stderr.

If you specify <*X*> and <*Y*> the **-verbose:gc** output is redirected to X files, each containing Y GC cycles.

The dump agent tokens can be used in the filename. See “Dump agent tokens” on page 234 for more information. If you do not specify <*file*>, `verbosegc.%Y%m%d.%H%M%S.%pid.txt` is used.

By default, no verbose GC logging occurs.

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 39. For information about when to use this policy, see "When to use the Balanced garbage collection policy" on page 43.

You specify the Balanced policy with 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 "Garbage Collector command-line options" on page 453.

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**
- **-Xloa**
- **-Xloainitial**<*percentage*>
- **-Xloamaximum**<*percentage*>
- **-Xloaminimum**<*percentage*>
- **-Xmo**<*size*>
- **-Xmoi**<*size*>
- **-Xmos**<*size*>
- **-Xmr**<*size*>
- **-Xmrx**<*size*>
- **-Xnoloa**
- **-Xnopartialcompactgc** (deprecated)
- **-Xpartialcompactgc** (deprecated)

A detailed description of these command-line options can be found in "Garbage Collector command-line options" on page 453.

JVM messages

Messages are issued by the IBM Java Virtual Machine (JVM) in response to certain conditions.

There are three main categories of message:

Information
Information messages provide information about JVM processing. For example, a dump information message is typically issued when a dump agent requests a Java dump.

Warning
Warning messages are issued by the JVM to indicate conditions that might need user intervention.

Error Error messages are issued by the JVM when normal processing cannot proceed, because of unexpected conditions.

IBM JVM messages have the following format:

```
JVMTYPENUM&
```

where:

- `JVM` is a standard prefix.
- `TYPE` refers to the JVM subcomponent that issued the message.
- `NUM` is a unique numerical number.
- `&` is one of the following codes:
  - `I` - Information message
  - `W` - Warning message
  - `E` - Error message

These messages can help you with problem determination. Refer to diagnostic information for more detailed information about diagnosing problems with the IBM JVM.

By default, all error and some information messages are routed to the system log and also written to `stderr` or `stdout`. The specific information messages are `JVMDUMP039I`, `JVMDUMP032I`, and `JVMDUMP033I`, which provide valuable additional information about dumps produced by the JVM. To route additional message types to the system log, or turn off message logging to the system log, use the **`-Xlog`** option. The **`-Xlog`** option does not affect messages written to the standard error stream (stderr). See "JVM command-line options" on page 428.

Finding logged messages

Logged messages can be found in different locations, according to platform.

Finding z/OS messages

On z/OS, messages are sent to the operator console. To see the messages, go from the **ispf** panel to the **sdsf** panel, then open the **log** panel.

Obtaining detailed message descriptions

Detailed message information is available to help with problem diagnosis.

Understanding the warning or error message issued by the JVM can help you diagnose problems. All warning and error messages issued by the JVM are listed by type in the IBM JVM Messages Guide.

- IBM JVM Messages

The messages, error codes, and exit codes in this guide apply to multiple versions of the JVM.

Note: If the JVM fills all available memory, the message number might be produced without a description for the error that caused the problem. Look for the

message number in the relevant section of the IBM JVM Messages Guide to see the message description and the additional information provided.

CORBA minor codes

This appendix gives definitions of the most common OMG- and IBM-defined CORBA system exception minor codes that the Java ORB from IBM uses.

See "Completion status and minor codes" on page 200 for more information about minor codes.

When an error occurs, you might find additional details in the ORB FFDC log. By default, the Java ORB from IBM creates an FFDC log with a filename in the format of `orbtrc.DDMMYYY.HHmm.SS.txt`. If the ORB is operating in the WebSphere Application Server or other IBM product, see the publications for that product to determine the location of the FFDC log.

CONN_CLOSE_REBIND CONN_CLOSE_REBIND

Explanation: An attempt has been made to write to a TCP/IP connection that is closing.

System action: org.omg.CORBA.COMM_FAILURE

User response: Ensure that the completion status that is associated with the minor code is **NO**, then reissue the request.

CONN_PURGE_ABORT CONN_PURGE_ABORT

Explanation: An unrecoverable error occurred on a TCP/IP connection. All outstanding requests are cancelled. Errors include:

- A GIOP MessageError or unknown message type
- An IOException that is received while data is being read from the socket
- An unexpected error or exception that occurs during message processing

System action: org.omg.CORBA.COMM_FAILURE

User response: Investigate each request and reissue if necessary. If the problem occurs again, enable ORB, network tracing, or both, to determine the cause of the failure.

CONNECT_FAILURE_1 CONNECT_FAILURE_1

Explanation: The client attempted to open a connection with the server, but failed. The reasons for the failure can be many; for example, the server might not be up or it might not be listening on that port. If a BindException is caught, it shows that the client could not open a socket locally (that is, the local port was in use or the client has no local address).

System action: org.omg.CORBA.TRANSIENT

User response: As with all TRANSIENT exceptions, trying again or restarting the client or server might solve the problem. Ensure that the port and server host names are correct, and that the server is running and allowing connections. Also ensure that no firewall is blocking the connection, and that a route is available between client and server.

CONNECT_FAILURE_5 CONNECT_FAILURE_5

Explanation: An attempt to connect to a server failed with both the direct and indirect IORs. Every client side handle to a server object (managed by the ClientDelegate reference) is set up with two IORs (object references) to reach the servant on the server. The first IOR is the direct IOR, which holds details of the server hosting the object. The second IOR is the indirect IOR, which holds a reference to a naming server that can be queried if the direct IOR fails.

Note: The two IORs might be the same at times. For any remote request, the ORB tries to reach the servant object using the direct IOR and then the indirect IOR. The CONNECT_FAILURE_5 exception is thrown when the ORB failed with both IORs.

System action: org.omg.CORBA.TRANSIENT (minor code E07)

User response: The cause of failure is typically connection-related, for example because of "connection refused" exceptions. Other CORBA exceptions such as NO_IMPLEMENT or OBJECT_NOT_EXIST might also be the root cause of the (E07) CORBA.TRANSIENT exception. An abstract of the root exception is logged in the description of the (E07) CORBA.TRANSIENT exception. Review the details of the exception, and take any further action that is necessary.

CREATE_LISTENER_FAILED CREATE_LISTENER_FAILED

Explanation: An exception occurred while a TCP/IP listener was being created.

System action: org.omg.CORBA.INTERNAL

User response: The details of the caught exception are written to the FFDC log. Review the details of the exception, and take any further action that is necessary.

LOCATE_UNKNOWN_OBJECT
LOCATE_UNKNOWN_OBJECT

Explanation: The server has no knowledge of the object for which the client has asked in a locate request.

System action: org.omg.CORBA.OBJECT_NOT_EXIST

User response: Ensure that the remote object that is requested resides in the specified server and that the remote reference is up-to-date.

NULL_PI_NAME NULL_PI_NAME

Explanation: One of the following methods has been called:

```
org.omg.PortableInterceptor.ORBInitInfoOperations.
add_ior_interceptor

org.omg.PortableInterceptor.ORBInitInfoOperations.
add_client_request_interceptor

org.omg.PortableInterceptor.ORBInitInfoOperations
.add_server_request_interceptor
```

The name() method of the interceptor input parameter returned a null string.

System action: org.omg.CORBA.BAD_PARAM

User response: Change the interceptor implementation so that the name() method returns a non-null string. The name attribute can be an empty string if the interceptor is anonymous, but it cannot be null.

ORB_CONNECT_ERROR_6
ORB_CONNECT_ERROR_6

Explanation: A servant failed to connect to a server-side ORB.

System action: org.omg.CORBA.OBJ_ADAPTER

User response: See the FFDC log for the cause of the problem, then try restarting the application.

POA_DISCARDING POA_DISCARDING

Explanation: The POA Manager at the server is in the discarding state. When a POA manager is in the discarding state, the associated POAs discard all incoming requests (for which processing has not yet begun). For more details, see the section that describes the POAManager Interface in the http://www.omg.org/cgi-bin/doc?formal/99-10-07.

System action: org.omg.CORBA.TRANSIENT

User response: Put the POA Manager into the active state if you want requests to be processed.

RESPONSE_INTERRUPTED
RESPONSE_INTERRUPTED

Explanation: The client has enabled the AllowUserInterrupt property and has called for an interrupt on a thread currently waiting for a reply from a remote method call.

System action: org.omg.CORBA.NO_RESPONSE

User response: None.

TRANS_NC_LIST_GOT_EXC
TRANS_NC_LIST_GOT_EXC

Explanation: An exception was caught in the NameService while the NamingContext.List() method was executing.

System action: org.omg.CORBA.INTERNAL

User response: The details of the caught exception are written to the FFDC log. Review the details of the original exception, and any further action that is necessary.

UNEXPECTED_CHECKED_EXCEPTION
UNEXPECTED_CHECKED_EXCEPTION

Explanation: An unexpected checked exception was caught during the servant_preinvoke method. This method is called before a locally optimized operation call is made to an object of type class. This exception does not occur if the ORB and any Portable Interceptor implementations are correctly installed. It might occur if, for example, a checked exception is added to the Request interceptor operations and these higher level interceptors are called from a back level ORB.

System action: org.omg.CORBA.UNKNOWN

User response: The details of the caught exception are written to the FFDC log. Check whether the class from which it was thrown is at the expected level.

UNSPECIFIED_MARSHAL_25
UNSPECIFIED_MARSHAL_25

Explanation: This error can occur at the server side while the server is reading a request, or at the client side while the client is reading a reply. Possible causes are that the data on the wire is corrupted, or the server and client ORB are not communicating correctly. Communication problems can caused when one of the ORBs has an incompatibility or bug that prevents it from conforming to specifications.

System action: org.omg.CORBA.MARSHAL

User response: Check whether the IIOP levels and CORBA versions of the client and server are compatible. Try disabling fragmentation (set com.ibm.CORBA.FragmentationSize to zero) to determine whether it is a fragmentation problem. In

this case, analysis of CommTraces (com.ibm.CORBA.CommTrace) might give extra information.

Environment variables

This appendix describes the use of environment variables. Environment variables are overridden by command-line arguments. Where possible, you should use command-line arguments rather than environment variables.

The following information about environment variables is provided:

- "Displaying the current environment"
- "Setting an environment variable"
- "Separating values in a list"
- "JVM environment settings" on page 470
- "z/OS environment variables" on page 473

Displaying the current environment

This description describes how to show the current environment and how to show an environment variable.

To show the current environment, run:

`set` (Windows)

`env` (UNIX)

`set` (z/OS)

`WRKENVVAR` (i5/OS command prompt)

`env` (i5/OS qsh or qp2term)

To show a particular environment variable, run:

`echo %ENVNAME%` (Windows)

`echo $ENVNAME` (UNIX, z/OS and I5/OS)

Use values exactly as shown in the documentation. The names of environment variables are case-sensitive in UNIX but not in Windows.

Setting an environment variable

This section describes how to set an environment variable and how long a variable remains set.

To set the environment variable **LOGIN_NAME** to *Fred*, run:

`set LOGIN_NAME=Fred` (Windows)

`export LOGIN_NAME=Fred` (UNIX ksh or bash shells and i5/OS)

These variables are set only for the current shell or command-line session.

Separating values in a list

The separator between values is dependant on the platform.

If the value of an environment variable is to be a list:

- On UNIX, i5/OS, and z/OS the separator is typically a colon (:).
- On Windows the separator is typically a semicolon (;).

JVM environment settings

This section describes common environment settings. The categories of settings are general options, deprecated JIT options, Javadump and Heapdump options, and diagnostic options.

General options

The following list summarizes common options. It is not a definitive guide to all the options. Also, the behavior of individual platforms might vary. See individual sections for a more complete description of behavior and availability of these variables.

CLASSPATH=<*directories and archive or compressed files*>
Set this variable to define the search path for application classes and resources. The variable can contain a list of directories for the JVM to find user class files and paths to individual Java archive or compressed files that contain class files; for example, `/mycode:/utils.jar` (UNIX or i5/OS), `D:\mycode;D:\utils.jar` (Windows).

Any class path that is set in this way is replaced by the **-cp** or **-classpath** Java argument if used.

IBM_JAVA_COMMAND_LINE
This variable is set by the JVM after it starts. Using this variable, you can find the command-line parameters set when the JVM started.

This setting is not available if the JVM is invoked using JNI.

IBM_JAVA_OPTIONS=<*option*>
Set this variable to store default Java options including **-X**, **-D** or **-verbose:gc** style options; for example, **-Xms256m -Djava.compiler**.

Any options set are overridden by equivalent options that are specified when Java is started.

This variable does not support **-fullversion** or **-version**.

If you specify the name of a trace output file either directly, or indirectly, using a properties file, the output file might be accidentally overwritten if you run utilities such as the trace formatter, dump extractor, or dump viewer. For information about avoiding this problem, see "Controlling the trace" on page 293, Note these restrictions.

JAVA_ASSISTIVE={ OFF | ON }
Set the **JAVA_ASSISTIVE** environment variable to OFF to prevent the JVM from loading Java Accessibility support.

JAVA_FONTS=<*list of directories*>
Set this environment variable to specify the font directory. Setting this variable is equivalent to setting the properties **java.awt.fonts** and **sun.java2d.fontpath**.

JAVA_PLUGIN_AGENT=<*version*>
Set this variable to specify the version of Mozilla.

This variable is for Linux and z/OS only.

JAVA_PLUGIN_REDIRECT=<*value*>
Set this variable to a non-null value to redirect JVM output, while serving as a plug-in, to files. The standard output is redirected to the file `plugin.out`. The error output is redirected to the file `plugin.err`.

This variable is for Linux and z/OS only.

LANG=<*locale*>
Set this variable to specify a locale to use by default.

This variable is for AIX, Linux, and z/OS only.

LIBPATH=<*list of directories*>
Set this variable to a colon-separated list of directories to define from where system and user libraries are loaded. You can change which versions of libraries are loaded, by modifying this list.

This variable is for AIX, i5/OS, and z/OS only.

SYS_LIBRARY_PATH=<*path*>
Set this variable to define the library path.

This variable is for Linux and z/OS only.

Deprecated JIT options

The following list describes deprecated JIT options:

IBM_MIXED_MODE_THRESHOLD
Use **-Xjit:count=**<*value*> instead of this variable.

JAVA_COMPILER
Use **-Djava.compiler=**<*value*> instead of this variable.

Javadump and Heapdump options

The following list describes the Javadump and Heapdump options. The recommended way of controlling the production of diagnostic data is the **-Xdump** command-line option, described in "Using dump agents" on page 221.

DISABLE_JAVADUMP={ TRUE | FALSE }
This variable disables Javadump creation when set to TRUE.

Use the command-line option **-Xdisablejavadump** instead. Avoid using this environment variable because it makes it more difficult to diagnose failures. On z/OS, use **JAVA_DUMP_OPTS** in preference.

IBM_HEAPDUMP or IBM_HEAP_DUMP={ TRUE | FALSE }
These variables control the generation of a Heapdump.

When the variables are set to 0 or FALSE, Heapdump is not available. When the variables are set to anything else, Heapdump is enabled for crashes or user signals. When the variables are not set, Heapdump is not enabled for crashes or user signals.

IBM_HEAPDUMP_OUTOFMEMORY={ TRUE | FALSE }
This variable controls the generation of a Heapdump when an out-of-memory exception is thrown.

When the variable is set to TRUE or 1 a Heapdump is generated each time an out-of-memory exception is thrown, even if it is handled. When the variable is set to FALSE or 0, a Heapdump is not generated for an out-of-memory exception. When the variable is not set, a Heapdump is generated when an out-of-memory exception is not caught and handled by the application.

IBM_HEAPDUMPDIR=<*directory*>
This variable specifies an alternative location for Heapdump files.

On z/OS, **_CEE_DMPTARG** is used instead.

IBM_JAVACOREDIR=<*directory*>
This variable specifies an alternative location for Javadump files; for example, on Linux `IBM_JAVACOREDIR=/dumps`

On z/OS, **_CEE_DMPTARG** is used instead.

IBM_JAVADUMP_OUTOFMEMORY={ TRUE | FALSE }
This variable controls the generation of a Javadump when an out-of-memory exception is thrown.

When the variable is set to TRUE or 1, a Javadump is generated each time an out-of-memory exception is thrown, even if it is handled. When the variable is set to FALSE or 0, a Javadump is not generated for an out-of-memory exception. When the variable is not set, a Javadump is generated when an out-of-memory exception is not caught and handled by the application.

IBM_NOSIGHANDLER={ TRUE }
This variable disables the signal handler when set to any value. If no value is supplied, the variable has no effect and the signal handler continues to work.

The variable is equivalent to the command-line option **-Xrs:all**

JAVA_DUMP_OPTS=<*value*>
This variable controls how diagnostic data are dumped.

For a fuller description of **JAVA_DUMP_OPTS** and variations for different platforms, see "Dump agent environment variables" on page 237.

TMPDIR=<*directory*>
This variable specifies an alternative temporary directory. This directory is used only when Javadumps and Heapdumps cannot be written to their target directories, or the current working directory.

This variable defaults to /tmp on Linux, z/OS, AIX, and i5/OS.

Diagnostic options

The following list describes the diagnostic options:

IBM_COREDIR=<*directory*>
Set this variable to specify an alternative location for system dumps and snap trace.

On z/OS, **_CEE_DMPTARG** is used instead for snap trace, and transaction dumps are written to TSO according to **JAVA_DUMP_TDUMP_PATTERN**.

On Linux, the dump is written to the OS specified directory, before being moved to the specified location.

IBM_JVM_DEBUG_PROG=<*debugger*>
Set this variable to start the JVM under the specified debugger.

This variable is for Linux only.

IBM_MALLOCTRACE=TRUE
Setting this variable to a non-null value lets you trace memory allocation in the JVM. You can use this variable with the **-Dcom.ibm.dbgmalloc=true** system property to trace native allocations from the Java classes.

This variable is equivalent to the command-line option **-Xcheck:memory**.

IBM_XE_COE_NAME=<*value*>
Set this variable to generate a system dump when the specified exception occurs. The value supplied is the package description of the exception; for example, `java/lang/InternalError`.

A Signal 11 is followed by a JVMXE message and then the JVM terminates.

JAVA_PLUGIN_TRACE=TRUE
When this variable is set to TRUE or 1, a Java plug-in trace is produced for the session when an application runs. Traces are produced from both the Java and Native layer.

By default, this variable is set to FALSE, so that a Java plug-in trace is not produced.

z/OS environment variables

This section describes the environment variables of the z/OS JVM.

IBM_JAVA_ABEND_ON_FAILURE=Y
Tells the Java launcher to mark the Task Control Block (TCB) with an abend code if the JVM fails to load or is terminated by an uncaught exception. By default, the Java launcher will not mark the TCB.

JAVA_DUMP_OPTS
See "Using Heapdump" on page 262 for details.

JAVA_DUMP_TDUMP_PATTERN=string
Result: The specified string is passed to IEATDUMP to use as the data/set name for the Transaction Dump. The default string is as follows:

For the 31-bit JVM:

```
%uid.JVM.TDUMP.%job.D%y%m%d.T%H%M%S
```

For the 64-bit JVM:

```
%uid.JVM.%job.D%y%m%d.T%H%M%S.X&DS
```

where `%uid` is found from the following C code fragment:

```
pwd = getpwuid(getuid());
pwd->pw_name;
```

For more information about the other variables, see "Dump agent tokens" on page 234.

You can use the output of the **-Xdump:what** option to see the default string for your operating system. For more information, see "Using dump agents on z/OS" on page 238.

JAVA_LOCAL_TIME
The z/OS JVM does not look at the offset part of the TZ environment variable and will therefore incorrectly show the local time. Where local time is not GMT, you can set the environment variable **JAVA_LOCAL_TIME** to display the correct local time as defined by TZ.

JAVA_THREAD_MODEL
JAVA_THREAD_MODEL can be defined as one of:

NATIVE
JVM uses the standard, POSIX-compliant thread model that is provided by the JVM. All threads are created as _MEDIUM_WEIGHT threads.

HEAVY
JVM uses the standard thread package, but all threads are created as _HEAVY_WEIGHT threads.

MEDIUM
Same as NATIVE.

NULL
Default case: Same as NATIVE/MEDIUM.

Related information:

"Using dump agents on z/OS" on page 238
Dump output is written to different files, depending on the type of the dump. File names include a time stamp. The z/OS platform has an additional dump type called CEEDUMP.

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 it 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
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

JVM setting	Default	Setting affected by
JIT debug options	Disabled	c
Java2D max size of fonts with algorithmic bold	14 point	e
Java2D use rendered bitmaps in scalable fonts	Enabled	e
Java2D freetype font rasterizing	Enabled	e
Java2D use AWT fonts	Disabled	e

JVM setting	AIX	IBM i	Linux	Windows	z/OS	Setting affected by
Default locale	None	None	None	N/A	None	e
Time to wait before starting plug-in	N/A	N/A	Zero	N/A	N/A	e
Temporary directory	`/tmp`	`/tmp`	`/tmp`	`c:\temp`	`/tmp`	e
Plug-in redirection	None	None	None	N/A	None	e
IM switching	Disabled	Disabled	Disabled	N/A	Disabled	e
IM modifiers	Disabled	Disabled	Disabled	N/A	Disabled	e
Thread model	N/A	N/A	N/A	N/A	Native	e
Initial stack size for Java Threads 32-bit. Use: **-Xiss<size>**	2 KB	2 KB	2 KB	2 KB	2 KB	c
Maximum stack size for Java Threads 32-bit. Use: **-Xss<size>**	256 KB	256 KB	256 KB	256 KB	256 KB	c
Stack size for OS Threads 32-bit. Use **-Xmso<size>**	256 KB	256 KB	256 KB	32 KB	256 KB	c
Initial stack size for Java Threads 64-bit. Use: **-Xiss<size>**	2 KB	N/A	2 KB	2 KB	2 KB	c
Maximum stack size for Java Threads 64-bit. Use: **-Xss<size>**	512 KB	N/A	512 KB	512 KB	512 KB	c
Stack size for OS Threads 64-bit. Use **-Xmso<size>**	256 KB	N/A	256 KB	256 KB	256 KB	c
Initial heap size. Use **-Xms<size>**	4 MB	4 MB	4 MB	4 MB	4 MB	c
Maximum Java heap size. Use **-Xmx<size>**	Half the available memory with a minimum of 16 MB and a maximum of 512 MB	2 GB	Half the available memory with a minimum of 16 MB and a maximum of 512 MB	Half the 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

JVM setting	AIX	IBM i	Linux	Windows	z/OS	Setting affected by
Compressed references	Disabled. From service refresh 4, enabled for **-Xmx** values of less than or equal to 25 GB	Disabled. From service refresh 4, enabled for **-Xmx** values of less than or equal to 25 GB	Disabled. From service refresh 4, enabled for **-Xmx** values of less than or equal to 25 GB	Disabled. From service refresh 4, enabled for **-Xmx** values of less than or equal to 25 GB	Disabled	ec
Page size for the Java object heap and code cache. For restrictions, see "Configuring large page memory allocation" on page 154	Operating system default	Operating system default	Architecture: • Linux on x86 and AMD64/ EM64T: 2 MB • Linux on System z: 1 MB • Other architectures: operating system default	Operating system default	1M pageable	c

Note: For earlier releases of Java, the value of **-Xmx** for the Windows JVM is half the physical memory with a minimum of 16 MB, and a maximum of 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 or limitations that you might encounter in specific system environments, or configurations.

If you find a problem, see the "Hints and Tips" pages, at http://www.ibm.com/systems/z/os/zos/tools/java/faq/javafaq.html.

If you find a problem that you have been unable to solve after looking through the "Hints and Tips" pages, see http://www.ibm.com/systems/z/os/zos/tools/java/services/services.html for advice and information about how to raise problems.

The problems described in this topic might not be limitations with the SDK or JRE. Instructions are provided to work around problems, where possible.

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
   ```

Java 2D rendering pipeline

The improved Java 2D graphics pipeline, based on the X11 XRender extension, accelerates rendering using hardware support. However, the XRender library is not supported on the z/OS operating system, and is therefore not available in the SDK. If the new pipeline is not present, Java 2D uses the existing X11 pipeline.

These known issues and limitations also apply to earlier releases of the SDK and JRE:

Limitation on class path length

If there are more than 2031 characters in your class path, the shell truncates your class path to 2031 characters. If you need a class path longer than 2031 characters, use the extension class loader option to refer to directories containing your `.jar` files, for example:

```
-Djava.ext.dirs=<directory>
```

Where *<directory>* is the directory containing your `.jar` files.

ThreadMXBean Thread User CPU Time limitation

There is no way to distinguish between user mode CPU time and system mode CPU time on this platform. ThreadMXBean.getThreadUserTime(), ThreadMXBean.getThreadCpuTime(), ThreadMXBean.getCurrentThreadUserTime(), and ThreadMXBean.getCurrentThreadCpuTime() all return the total CPU time for the required thread.

You can get the CPU time only for the current thread by calling ThreadMXBean.isCurrentThreadCpuTimeSupported(). Calling ThreadMXBean.isThreadCpuTimeSupported() returns a value of false because getting the CPU time for a thread other than the current thread is not supported.

Using -Xshareclasses:destroy during JVM startup

When running the command **`java -Xshareclasses:destroy`** on a shared cache that is being used by a second JVM during startup, you might have the following issues:

- The second JVM fails.

- The shared cache is deleted.

Changes to locale translation files

From Java 7 service refresh 1, changes are made to the locale translation files to make them consistent with Oracle JDK 7. The same changes were applied to the IBM SDK for Java 6 for consistency with Oracle JDK 6. To understand the differences in detail, see this Java 6 support document:http://www.ibm.com/support/docview.wss?uid=swg21568667.

Large page request fails

There is no error message issued when the JVM is unable to honor the **-Xlp** request.

There are a number of reasons why the JVM cannot honor a large page request. For example, there might be insufficient large pages available on the system at the time of the request. To check whether the **-Xlp** request was honored, you can review the output from **-verbose:gc**. Look for the attributes `requestedPageSize` and `pageSize` in the **-verbose:gc** log file. The attribute `requestedPageSize` contains the value specified by **-Xlp**. The attribute `pageSize` is the actual page size used by the JVM.

ThreadMXBean thread CPU time might not be monotonic on SMP systems

On SMP systems, the times returned by ThreadMXBean.getThreadUserTime(), ThreadMXBean.getThreadCpuTime(), ThreadMXBean.getCurrentThreadUserTime(), and ThreadMXBean.getCurrentThreadCpuTime() might not increase monotonically if the relevant thread migrates to a different processor.

Unexpected `CertificateException`

IBM SDK for z/OS, V7 service refresh 4 fix pack 1 and later releases contain a security enhancement to correctly validate certificates on jar files of applications. After upgrading, a `CertificateException` occurs for any applications in one of the following scenarios:

- The application jar is not properly signed.
- The application jar has incorrect certificates.
- A certificate in the certificate chain is revoked.

To avoid these exceptions, make sure that your application jars are signed with valid certificates before you upgrade from an earlier release. This issue relates to APAR IV38456.

Unexpected application errors with RMI

If your application uses RMI and you experience unexpected errors after applying IBM SDK for z/OS, V7 service refresh 4 fix pack 2, or later releases, the problem might be associated with a change to the default value of the RMI property java.rmi.server.useCodebaseOnly. For more information, see http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/enhancements-7.html.

Unexpected XSLT error on extension elements or extension functions when Java security is enabled

From IBM SDK for z/OS, V7 service refresh 5, any attempt to use extension elements or extension functions when Java security is enabled, results in a `javax.xml.transform.TransformerException` error during XSLT processing. This change in behavior is introduced to enhance security.

The following XSLT message is generated when extension functions are used: `Use of the extension function '<method name>' is not allowed when Java security is enabled. To override this, set the com.ibm.xtq.processor.overrideSecureProcessing property to true. This override only affects XSLT processing.`

The following XSLT message is generated when extension elements are used: `Use of the extension element '<element name>' is not allowed when Java security is enabled. To override this, set the com.ibm.xtq.processor.overrideSecureProcessing property to true. This override only affects XSLT processing.`

To allow extensions when Java security is enabled, set the **`com.ibm.xtq.processor.overrideSecureProcessing`** system property to *true*. For more information about this system property, see "-Dcom.ibm.xtq.processor.overrideSecureProcessing" on page 421.

Behavior change to java.lang.logging.Logger

To enhance security, java.lang.logging.Logger no longer walks the stack to search for resource bundles. Instead, the resource bundles are located by using the caller 's class loader. If your application depends upon stack-walking to locate resource bundles, this behavior change might affect your application. To work around this problem, a system property is available in this release to revert to the earlier behavior. To set this property on the command line specify: **`-Djdk.logging.allowStackWalkSearch=true`**.

Support for virtualization software

The IBM SDK for Java is tested with a number of virtualized server products.

The following virtualization software is tested with the latest release of the IBM SDK for z/OS, V7:

Table 12. Virtualization software tested for the IBM SDK for Java

Vendor	Architecture	Server virtualization	Version
IBM	System z	PR/SM™	z10, z11, z196, zEC12
IBM	System z	z/VM®	6.1, 6.2
IBM	POWER®	PowerVM® Hypervisor	Power6, Power7
VMware	x86-64	VMware ESX and ESXi Server	4.1, 5.0
Red Hat	x86-64	Red Hat Enterprise Virtualization (RHEV)	2.1, 3.0
SUSE	x86-64	SUSE KVM	SLES 11

Table 12. Virtualization software tested for the IBM SDK for Java (continued)

Vendor	Architecture	Server virtualization	Version
Microsoft	x86-64	Hyper-V	Server 2012

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1758
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web

sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

JIMMAIL@uk.ibm.com

[Hursley Java Technology Center (JTC) contact]

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided “AS IS”, without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Privacy Policy Considerations

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.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at Copyright and trademark information.

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.

Other company, product and service names may be trademarks or service marks of others.

Printed in USA

ZSL03227-USEN-02