IBM for AIX, Java Technology Edition
Version 7

SDK and Runtime Guide

IBM for AIX, 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 495.

Copyright information

This edition of the user guide applies to the IBM SDK for AIX, Java Technology Edition, Version 7, and to all subsequent releases, modifications, and Service Refreshes, until otherwise indicated in new editions.

© Copyright 1997, 2011, Oracle and/or its affiliates. All rights reserved.

© Copyright IBM Corporation 2011.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Preface vii

Chapter 1. Product overview 1
Introduction to Java 1
IBM Software Developers Kit (SDK) 1
IBM Java Runtime Environment (JRE) 4
IBM Java Virtual Machine (JVM). 5
Conventions and terminology 6
What's new 6
IBM J9 Virtual Machine 7
Memory management 7
Class data sharing 8
The JIT compiler 9
Diagnostics component 9
Security component 10
Real time evaluation technology 10
Other sources of information 17
Accessibility 17
Reporting problems 17

Chapter 2. Understanding the IBM Software Developers Kit (SDK) for Java 19
The building blocks of the IBM Virtual Machine for Java 19
Java application stack 20
Components of the IBM Virtual Machine for Java 21
Memory management 23
Overview of memory management 23
Allocation 26
Detailed description of global garbage collection 29
Generational Concurrent Garbage Collector. . . 37
Balanced Garbage Collection policy 39
How to do heap sizing 45
Interaction of the Garbage Collector with applications 47
How to coexist with the Garbage Collector . . . 47
Frequently asked questions about the Garbage Collector 50
Class loading 53
The parent-delegation model 53
Namespaces and the runtime package 54
Custom class loaders 54
Class data sharing 55
The JIT compiler. 56
JIT compiler overview 57
How the JIT compiler optimizes code. 57
Frequently asked questions about the JIT compliler 59
The AOT compiler 60
Java Remote Method Invocation 61
The RMI implementation 61
Thread pooling for RMI connection handlers . . 62
Understanding distributed garbage collection . . 63
Debugging applications involving RMI 63
The ORB 64
CORBA. 64
RMI and RMI-IIOP 65
Java IDL or RMI-IIOP? 65
RMI-IIOP limitations 65
Further reading 66
Examples of client–server applications 66
Using the ORB 71
How the ORB works 75
Additional features of the ORB 82
The Java Native Interface (JNI) 88
Overview of JNI 88
The JNI and the Garbage Collector 89
Copying and pinning 94
Handling exceptions 96
Synchronization 96
Debugging the JNI 97
JNI checklist 98

Chapter 3. Planning. 101
Migrating from other IBM JVMs 101
Version compatibility 101
Supported environments 101
Additional information for AIX 102

Chapter 4. Installing and configuring the SDK and Runtime Environment . . 105
Installing from an InstallAnywhere package . . . 106
Completing an attended installation 106
Completing an unattended installation 107
Interrupted installation 108
Known issues and limitations 108
Relocating an installp package. 109
Upgrading the SDK 109
Verification 109
Setting the path 110
Setting the class path 111
Updating your SDK or JRE for daylight saving time changes 111

Chapter 5. Developing Java applications 113
Using XML 113
Migrating to the XL-TXE-J 115
XML reference information 117
Debugging Java applications 122
Java Debugger (JDB) 122
Determining whether your application is running on a 32-bit or 64-bit JVM 123
How the JVM processes signals 123
Signals used by the JVM 124
Linking a native code driver to the signal-chaining library 126
Writing JNI applications 127
Supported compilers 127
JNI compatibility 127

© Copyright IBM Corp. 2011

Example of using AIX shared libraries 130
Support for thread-level recovery of blocked connectors 131
CORBA support 132
System properties for tracing the ORB 133
System properties for tuning the ORB 134
Java security permissions for the ORB 134
ORB implementation classes 135
RMI over IIOP 135
RMI-IIOP Programmer's Guide 135
Implementing the Connection Handler Pool for RMI 142
Enhanced BigDecimal 143
AIX native threads 143
JNDI 143
HPROF Performance Profiler and JVMPI 143
GetCurrentThreadCpuTime. 143
Known restriction 144
Support for XToolkit 144
Support for the Java Attach API 145
Plug-in, Applet Viewer and Web Start 147
Using the Java plug-in 147
Working with applets 148
Using Web Start 149
Distributing Java applications 151
Java Communications API (JavaComm) 151
Location of the Java Communications API files 152
Mapping serial ports 152
Configuring the Java Communications API . . 152
Printing limitation with the Java Communications API. 153
The Java Communications API documentation 153

Chapter 6. Running Java applications 155

The **java** and **javaw** commands 155
Obtaining version information. 155
Specifying Java options and system properties 156
Standard options 156
Globalization of the **java** command 158
Working with the **LIBPATH** environment variable 158
Working with the **LDR_CNTRL** environment variable 159
The Just-In-Time (JIT) compiler 160
Disabling the JIT 161
Enabling the JIT 161
Determining whether the JIT is enabled . . . 161
Specifying garbage collection policy 162
Garbage collection options 162
More effective heap usage using compressed references 163
Pause time 163
Pause time reduction 163
Environments with very full heaps 164
Dynamic Logical Partitioning (DLPAR) support 165
Live application mobility on AIX WPAR 166
Using the IPv6 extensions 166
Enhanced BiDirectional support 166
Euro symbol support 167
Scaling support. 167
SDK use of AIX large program support. 167
Automatic LDR_CNTRL=MAXDATA values . . 168
Manual LDR_CNTRL=MAXDATA values . . . 170
AIX Stack Execution Disable 170

Chapter 7. Performance 173

Class data sharing between JVMs 173
Overview of class data sharing 173
Class data sharing command-line options . . . 175
Creating, populating, monitoring, and deleting a cache 179
Performance and memory consumption . . . 180
Considerations and limitations of using class data sharing. 181
Adapting custom class loaders to share classes 183
Performance problems 183

Chapter 8. Security Guide 185

General information about IBM security providers 185
Hardware Cryptographic support 185
IBM PCI-X Cryptographic Coprocessor (PCIXCC). 186
iKeyman tool 186
Java Authentication and Authorization Service (JAAS) V2.0 187
History of changes 187
Java Certification Path (CertPath). 189
History of changes 189
Java Cryptography Extension (JCE) 191
History of changes 193
Known limitations. 193
Java Generic Security Service (JGSS). 194
Installing Kerberos 194
History of changes 195
IBMJSSE2 Provider 197
Differences between the IBMJSSE Provider and the IBMJSSE2 Provider 197
Differences between the IBMJSSE2 Provider and the Oracle version of JSSE 198
History of changes 199
IBMJSSE2 support for RFC 5746 Transport Layer Security (TLS) – Renegotiation Indication Extension. 201
IBMPKCS11Impl Provider 203
History of changes 204
IBMJCEFIPS Provider 205
IBM SASL Provider 206
IBM Key Certificate Management. 207
Keytool 208
Java XML Encryption and signatures 208
IBM SecureRandom Provider 209

Chapter 9. Troubleshooting and support 211

Submitting problem reports 211
Problem determination 211
First steps in problem determination 212
AIX problem determination 213
ORB problem determination 246
NLS problem determination 259
Attach API problem determination 261
Using diagnostic tools 263
Overview of the available diagnostics 264

Using dump agents 271
Using Javadump 286
Using Heapdump 306
Using system dumps and the dump viewer . . 315
Tracing Java applications and the JVM 331
JIT and AOT problem determination 364
The Diagnostics Collector 370
Garbage Collector diagnostics 375
Class-loader diagnostics 383
Shared classes diagnostics 385
Using the Reliability, Availability, and Serviceability Interface 412
Using the HPROF Profiler 425
Using the JVMTI 429
Using the Diagnostic Tool Framework for Java 442
Using JConsole 449
Using the IBM Monitoring and Diagnostic Tools for Java - Health Center 454

Chapter 10. Reference 455
Command-line options 455
Specifying command-line options. 455
General command-line options 456
System property command-line options . . . 457
JVM command-line options. 459
JIT and AOT command-line options 471
Garbage Collector command-line options . . . 474
Messages 482
Finding logged messages 483
Obtaining detailed message descriptions . . . 484
CORBA minor codes 484
Environment variables 486
Displaying the current environment 486
Setting an environment variable 487
Separating values in a list 487
JVM environment settings 487
Default settings for the JVM 490
Known issues and limitations 492
Support for virtualization software 493

Notices 495
Trademarks 497

Preface

This guide provides general information about the IBM® SDK and Runtime Environment for AIX®, Java Technology Edition, Version 7. The guide gives specific information about any differences in the IBM implementation compared with the Oracle implementation.

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

Useful Web sites include:

- The Java download site for AIX
- IBM home page for Java

This guide is part of a release and is applicable only to that particular release. Make sure that you have the guide appropriate to the release you are using.

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

Technical changes made for this version of the guide, other than minor or obvious ones, are indicated by blue chevrons when viewing in an Information Center, by blue chevrons and in red when viewing in HTML, or by vertical bars to the left of the changes when viewing as a PDF file.

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.

© Copyright IBM Corp. 2011

Chapter 1. Product overview

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

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 supplied by IBM, 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/java7[_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.

© Copyright IBM Corp. 2011

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.

ControlPanel (Java Control Panel)
Configures your Runtime Environment.

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

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

ikeycmd (iKeyman command-line utility)
Allows you to manage keys, certificates, and certificate requests from the command line. For more information see Chapter 8, "Security Guide," on page 185.

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.

javaws (Java Web Start)
Enables the deployment and automatic maintenance of Java applications. For more information, see "Running Web Start" on page 150.

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

jdb (Java Debugger)
Helps debug your Java programs.

jdmpview (Cross-platform dump formatter)
Analyzes dumps. For more information, see the "Problem determination" on page 211.

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

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

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

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

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

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

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

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

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

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

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

xjc
Compiles XML Schema files.

Include Files
C headers for JNI programs.

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 AIX software.

License file

The License file, /usr/swlag/<*locale*>/Java6_64.la, contains the license agreement for the SDK for AIX software (where <*locale*> is the name of your locale, for example en). To view or print the license agreement, open the file in a Web browser.

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/java7[_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

ikeyman (iKeyman GUI utility)
Allows you to manage keys, certificates, and certificate requests. For more information see the accompanying Security Guide and http://public.dhe.ibm.com/software/dw/jdk/security/50/GSK7c_SSL_IKM_Guide.pdf. The SDK also provides a command-line version of this utility.

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.

javaws (Java Web Start)
Enables the deployment and automatic maintenance of Java applications. For more information, see "Running Web Start" on page 150.

jcontrol (Java Control Panel)
Configures your Runtime Environment.

jextract (Dump extractor)
Converts a system-produced dump into a common format that can be used by **jdmpview**. For more information, see "Using jextract" on page 318.

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)
- Diagnostics 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...`

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**][<*suboptions*>=<*value*>...]

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

What's new

This topic introduces new material for Version 7.

Technical changes made for this documentation, other than minor or obvious ones, are indicated as follows:

- Blue chevrons when viewing in an Information Center.
- Vertical bars to the left of the changes when viewing as a PDF file.

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

Installing

You now install the SDK and runtime environment from InstallAnywhere packages, which use the .bin format. These packages replace the .rpm and .tgz packages that were previously provided. There are two types of InstallAnywhere package, installable and archive. The installable packages are equivalent to the

`.rpm` packages that were previously provided. The archive packages are equivalent to the `.tgz` packages that were previously provided. For more information, see Chapter 4, “Installing and configuring the SDK and Runtime Environment,” on page 105, and subtopics.

IBM J9 Virtual Machine

IBM SDK for AIX, v7 includes the IBM J9 V2.6 virtual machine (JVM). This version of the JVM has optimizations that are intended to improve performance.

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 245.

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 145.

Memory management

Changes to memory management for IBM SDK for AIX, v7 include a new garbage collection policy, and a redesign of verbose garbage collection logging to improve problem determination.

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 162.

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 375.

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 “Garbage Collector command-line options” on page 474.

Class data sharing

Class data sharing provides a method of reducing memory footprint and improving JVM startup time. There are a number of changes to class data sharing for IBM SDK for AIX, v7.

The following sections describe the changes.

.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 459.

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 388.

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 388.

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 274.

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 401.

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 432.

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 403.

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 459.

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 57.

Diagnostics component

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

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 333.

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 291.

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 442.

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 437.

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 301.

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 459.

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 430.

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 282. If you want to disable this new functionality, remove the new dump agent. For more information, see "Removing dump agents" on page 283.

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 310.

Security component

Changes include support for Transport Layer Security (TLS) V1.2 and Elliptic-curve cryptography.

The IBM security providers included with the IBM SDK for AIX, v7 are updated for this release. To read about the changes to each security component, see the Chapter 8, "Security Guide," on page 185.

Real time evaluation technology

The IBM Metronome Garbage Collector, a key component of the IBM WebSphere® Real Time product, is included with IBM SDK for AIX, v7 for evaluation.

Metronome Garbage Collector

The Metronome Garbage Collector (GC) is an incremental, deterministic garbage collector with short pause times. Applications that are dependent on precise response times can take advantage of this technology by avoiding potentially long delays from garbage collection activity. You activate the Metronome GC by specifying **-Xgcpolicy:metronome** on the command line.

Further information about using this garbage collector can be found in the following sections.

Hardware and software prerequisites

The Metronome Garbage Collector is supported on the same hardware and software combinations that are described in the topic "Supported environments" on page 101.

Considerations

You must be aware of a number of factors when using this technology:

- Where possible, do not run more than one real-time JVM on the same system. The reason is that you would then have multiple garbage collectors. Each JVM does not know about the memory areas of the other. Therefore, the garbage collectors might not have all the information for optimal results. If you must use multiple JVMs, ensure that each JVM is bound to a specific subset of processors by using the **execrset** command.
- When using shared class caches, the cache name must not exceed 53 characters.
- Workload partitions (WPAR) are not supported.
- Micropartitions are not supported. Logical partitions (LPAR) with an integral number of processors are supported, but LPARs with a fractional number of processors, for example 0.5 or 1.5, are not.

Performance on certified hardware configurations

Certified systems have sufficient clock granularity and processor speed to support performance goals. For example, a well-written application running on a system that is not overloaded, and with an adequate heap size, would normally experience GC pause times of about 3 milliseconds, and no more than 3.2 milliseconds. During GC cycles, an application with default environment settings is not paused for more than 30% of elapsed time during any sliding 60 millisecond window. The collective time spent in GC pauses over any 60 millisecond period should add up to no more than 18 milliseconds.

Compressed references

The Metronome GC supports both compressed and uncompressed references on 64-bit platforms. When using compressed references, the JVM stores all references to objects, classes, threads, and monitors as 32-bit values. Using compressed references improves the performance of many applications because objects are smaller, resulting in less frequent garbage collection and improved memory cache utilization. For further information about compressed references, see “Compressed references” on page 28.

IBM WebSphere Real Time product

The IBM WebSphere Real Time product contains further features and capabilities to support real-time application requirements. For more information about the IBM WebSphere Real Time product, see http://www-01.ibm.com/software/webservers/realtime/index.html.

Using the Metronome Garbage Collector

The Metronome Garbage Collector can be configured to optimize garbage collection for your environment.

Controlling processor utilization:

You can limit the amount of processing power available to the metronome garbage collector.

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

For example:

```
java -Xgcpolicy:metronome -Xgc:targetUtilization=80 yourApplication
```

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

Controlling pause time:

Metronome garbage collector (GC) pause time can be fine-tuned for each Java process.

By default, the Metronome GC pauses for 3 milliseconds in each individual pause, which is known as a quantum. A full garbage collection cycle requires many of these pauses, which are scattered so that the application has enough time to run. You can change this maximum individual pause time value with the **-Xgc:targetPauseTime** option. For example, running with **-Xgc:targetPauseTime=20** causes the GC to operate with individual pauses that are no longer than 20 milliseconds.

The IBM Monitoring and Diagnostics Tools for Java - Garbage Collection and Memory Visualizer (GCMV) can be used to monitor the GC pause times for your application, as well as helping to diagnose and tune performance problems in your Java application. The tool parses and plots data from various types of log, including:

- 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 graphs in this section are generated by GCMV, and show the affect of changing the target pause time on garbage collection cycles. Each graph plots the actual pause times between metronome garbage collection cycles (Y-axis) against the run time of an application (X-axis).

Note: GCMV supports the verbose garbage collection format available with earlier releases of the SDK. If you want to analyze verbose GC output with GCMV, generate the output with the **-Xgc:verboseFormat=deprecated** option. For more information, see "Garbage Collector command-line options" on page 474.

With the default target pause time set, the Verbose GC pause time graph shows that pause times are held around or below the 3 millisecond mark:

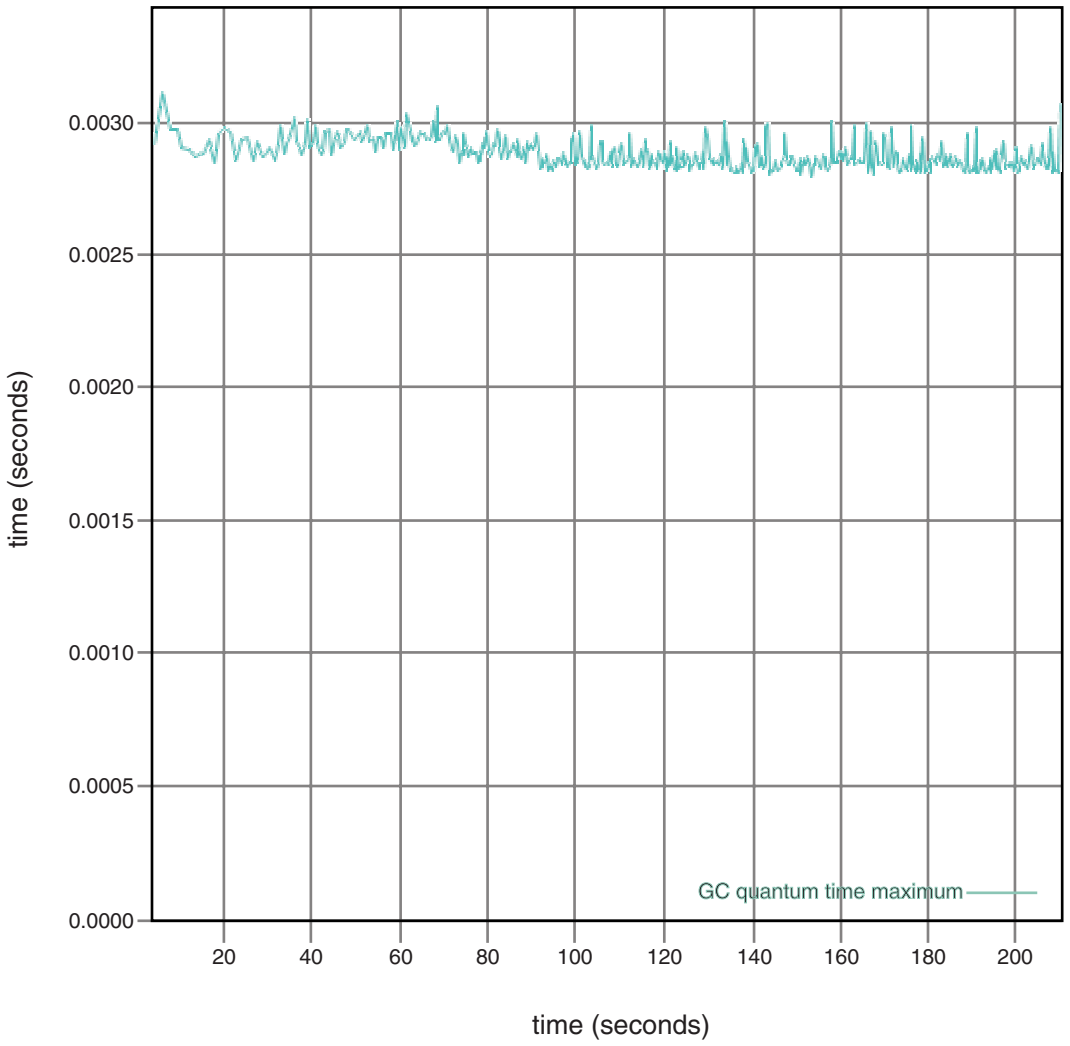


Figure 1. Actual garbage collection pause times when the target pause time is set to the default (3 milliseconds)

With a target pause time set at 6 milliseconds, the Verbose GC pause time graph shows that pause times are held around or below the 6 millisecond mark:

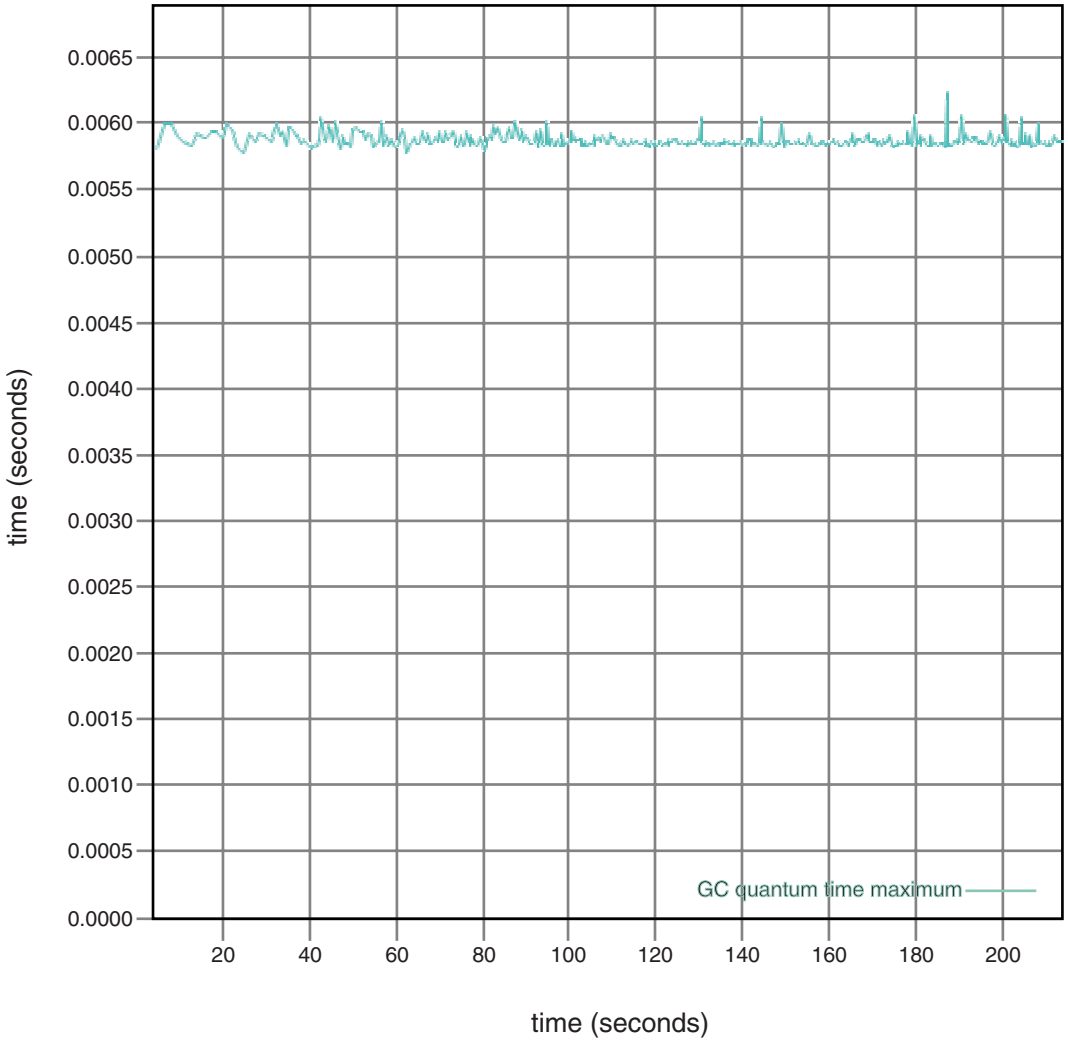


Figure 2. Actual pause times when the target pause time is set to 6 milliseconds

With a target pause time set at 10 milliseconds, the Verbose GC pause time graph shows that pause times are held around or below the 10 millisecond mark:

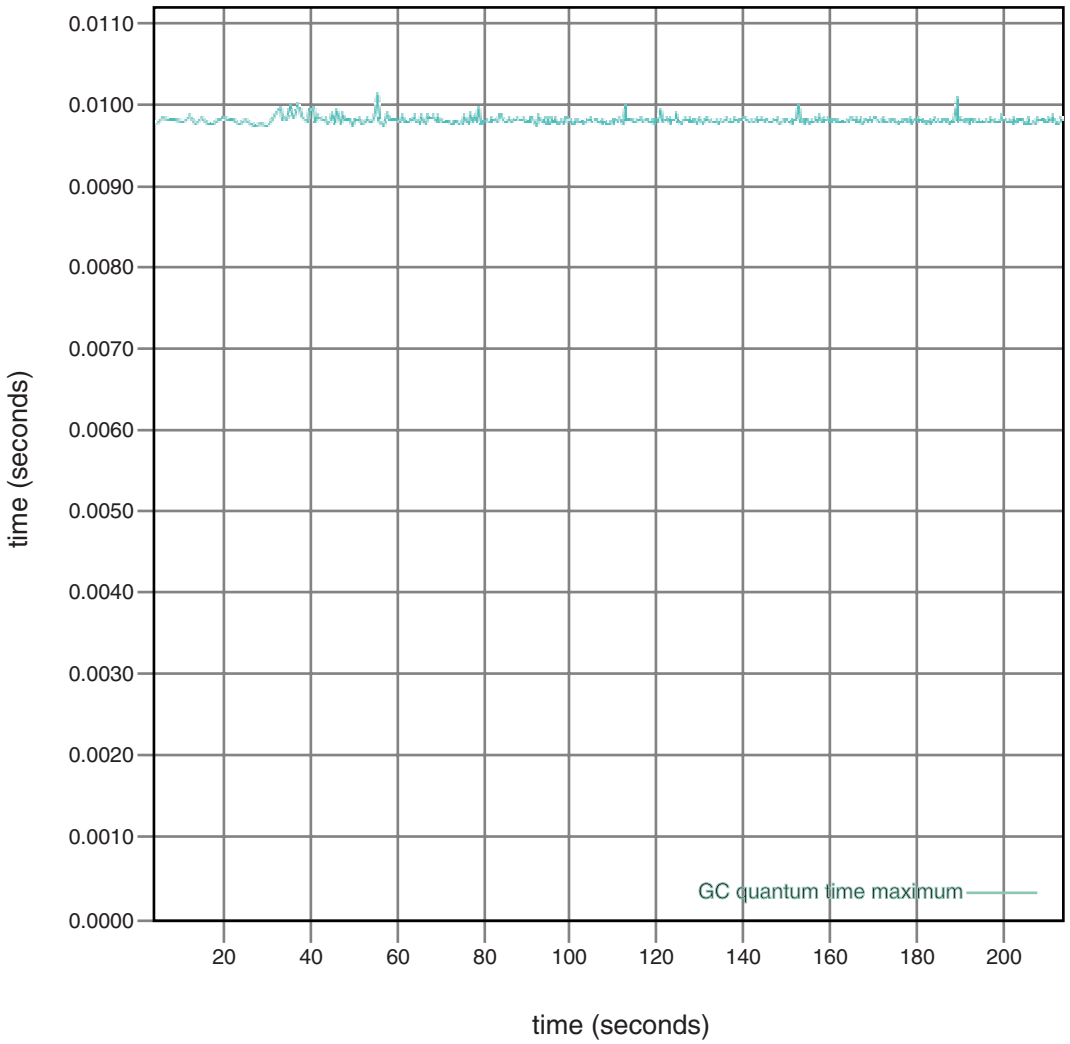


Figure 3. Actual pause times when the target pause time is set to 10 milliseconds

With a target pause time set at 15 milliseconds, the Verbose GC pause time graph shows that pause times are held around or below the 15 millisecond mark:

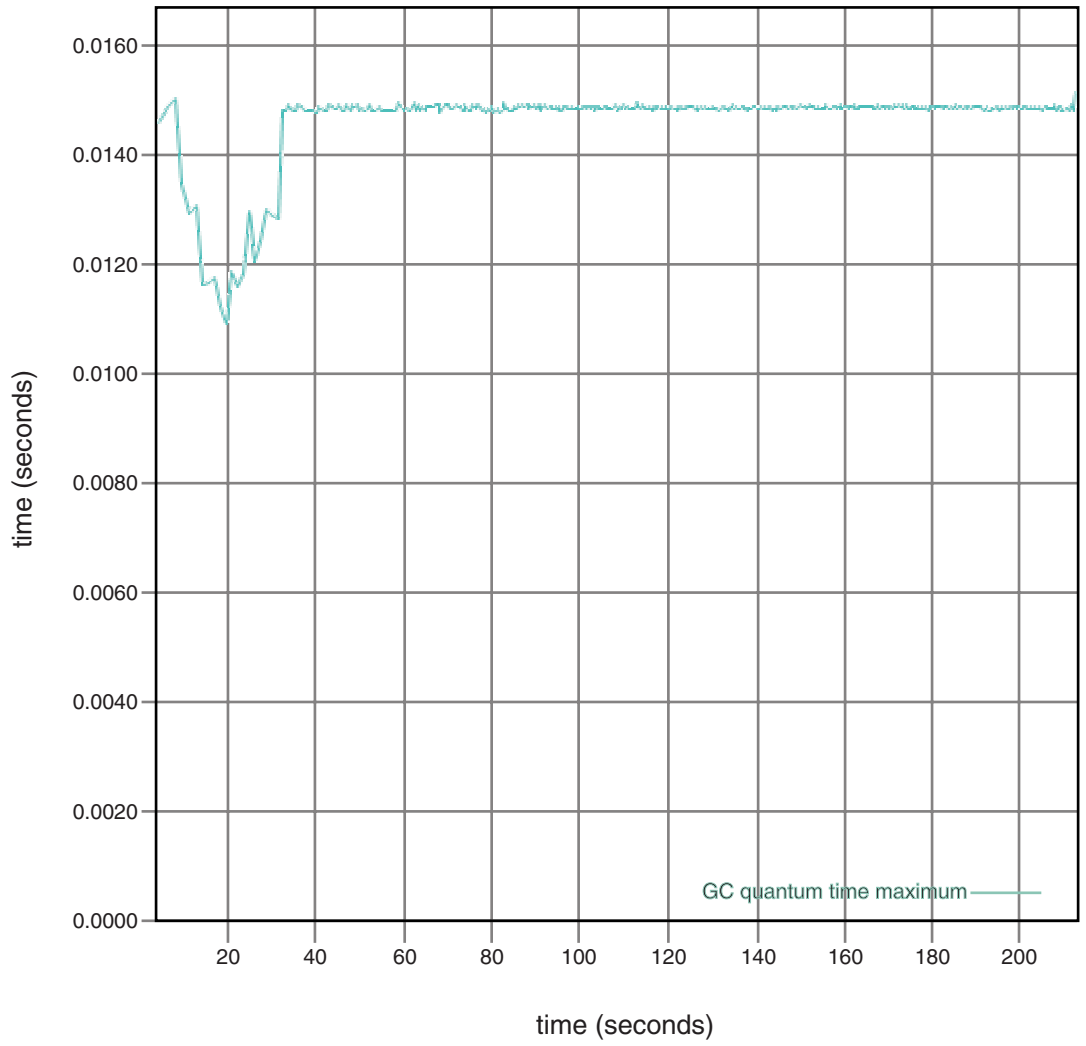


Figure 4. Actual pause times when the target pause time is set to 15 milliseconds

Metronome Garbage Collector limitation:

Under rare circumstances, you might experience longer than expected pauses during garbage collection.

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

- Live reference variables in the active thread call stacks.
- Static references.

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

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

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 Java 7 documentation as an Eclipse plug-in, or in PDF format for printing, see:

  http://www.ibm.com/developerworks/java/jdk/docs.html
- For Javadoc HTML documentation that has been generated from IBM SDK for Java APIs, see:

  http://www.ibm.com/developerworks/java/jdk/docs.html
- 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

The documentation has been tested using screen readers.

To change the font sizes use the function that is supplied with your browser, typically found under the **View** menu option.

For users who require keyboard navigation, a description of useful keystrokes for Swing applications is in *Swing Key Bindings* at http://www.ibm.com/developerworks/java/jdk/additional/.

Issues that affect accessibility are documented in “Known issues and limitations” on page 492.

Reporting problems

Use the problem determination section to help diagnose your problem, and learn about available workarounds. If you need to contact IBM service, you might need to collect some data.

The problem determination section provides guidance on diagnosing and correcting problems, including known workarounds. See “Problem determination” on page 211. If you cannot resolve the issue on your own, this section also tells you what data IBM service needs you to collect. Collect the data and send a problem report and associated data to IBM service, as described in “Submitting problem reports” on page 211.

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

The information in this section of the *Information Center* will give you a basic understanding of the SDK.

The content provides:

- Background information to explain why some diagnostics 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"
- "Memory management" on page 23
- "Class loading" on page 53
- "Class data sharing" on page 55
- "The JIT compiler" on page 56
- "Java Remote Method Invocation" on page 61
- "The ORB" on page 64
- "The Java Native Interface (JNI)" on page 88

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

© Copyright IBM Corp. 2011

- 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.

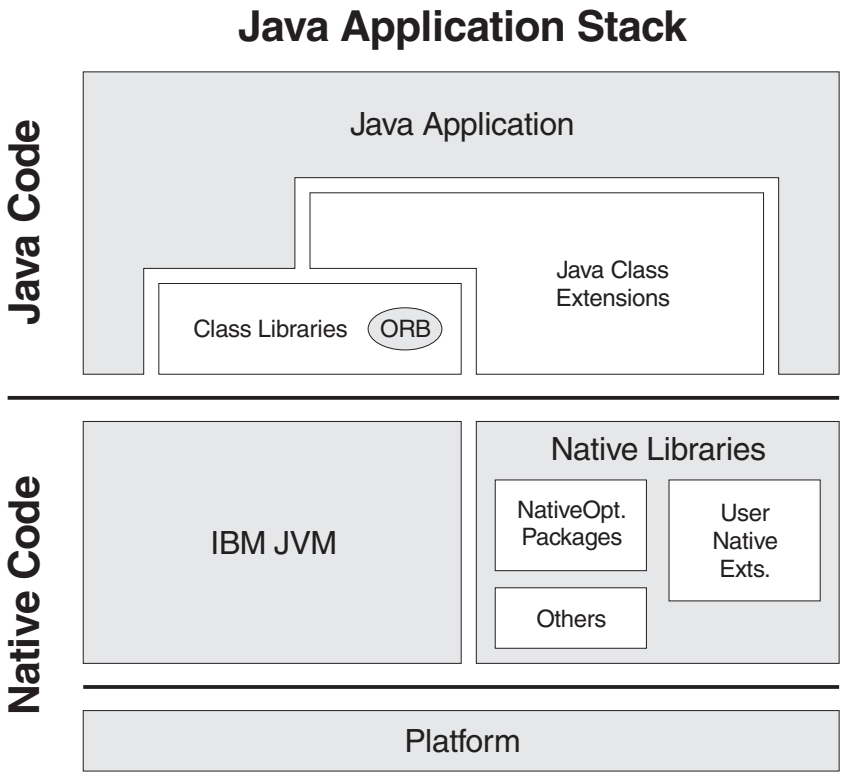
All the JVMs that are currently used commercially come with a supplementary compiler that takes bytecodes and produces platform-dependent machine code. This compiler works with the JVM to select parts of the Java program that could benefit from the compilation of bytecode, and replaces the JVM's virtualized 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 56.

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.

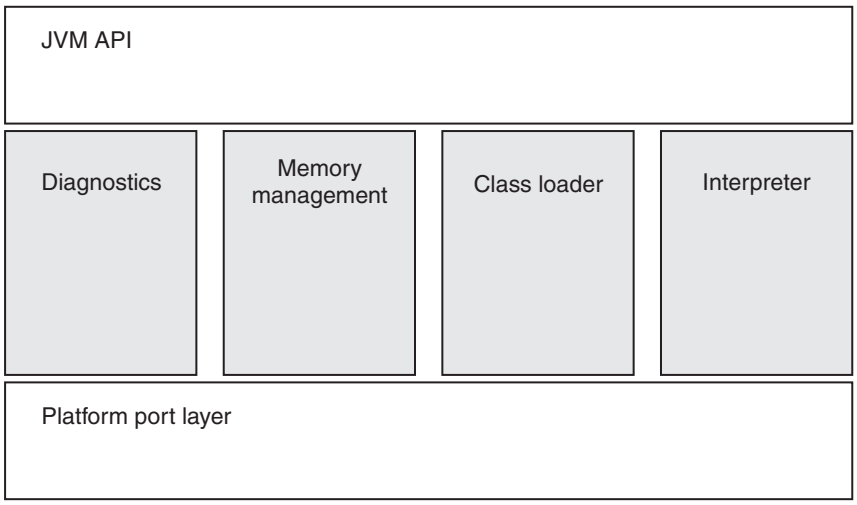


The JVM facilitates the invocation of native functions by Java applications and a number of well-defined Java Native Interface functions for manipulating Java from native code (for more information, see "The Java Native Interface (JNI)" on page 88).

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.

Diagnostics component

The diagnostics 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 diagnostics component to trace the runtime function of the JVM and help to identify the cause of the problem. The diagnostics component can produce output selectively from various parts of the JVM and the JIT. “Using diagnostic tools” on page 263 describes various uses of the diagnostics 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 56.

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 47
- "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 diagnostics" on page 375.

See also the reference information in "Garbage Collector command-line options" on page 474.

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.

There are additional considerations to these allocation rules when using the Balanced Garbage Collection policy. On x86 and POWER® architectures that have Non-Uniform Memory Architecture (NUMA) characteristics, allocation includes the segregation of threads across NUMA nodes to optimize memory access. For further information about the NUMA awareness available with Balanced Garbage Collection, see "NUMA awareness" on page 40.

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.
: Try using **-Xgcpolicy:optavgpause**. This 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.

  Try using **-Xgcpolicy:balanced** if pause times are unacceptable during a global garbage collection, due to a large heap size. 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.

For a description of cache allocation, when the request cannot be satisfied, see "Cache allocation." As its name implies, heap lock allocation requires a lock and is therefore avoided, if possible, by using the cache.

If the Garbage Collector cannot find a big enough chunk of free storage, allocation fails and the Garbage Collector must perform 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 has been 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** command-line option to enable compressed references in a 64-bit JVM. Only 64-bit JVMs recognize this option.

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 the option to determine if it is appropriate.

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 317.

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.

From Java 6 SR7, 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.

For Java 6 SR 5 and after, 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 Requests the combined use of concurrent and generational GC to help minimize the time that is spent in any garbage collection pause. **gencon** is the default setting.

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 above 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 is 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 below 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 474. 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 be left 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 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 going below 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 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.

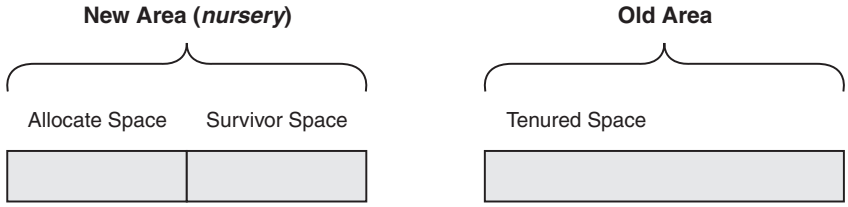
Where paging is supported, the GC allocates physical memory to the initial heap to the amount that is specified by the **-Xms** parameter. Additional memory is committed as the heap grows.

Generational Concurrent Garbage Collector

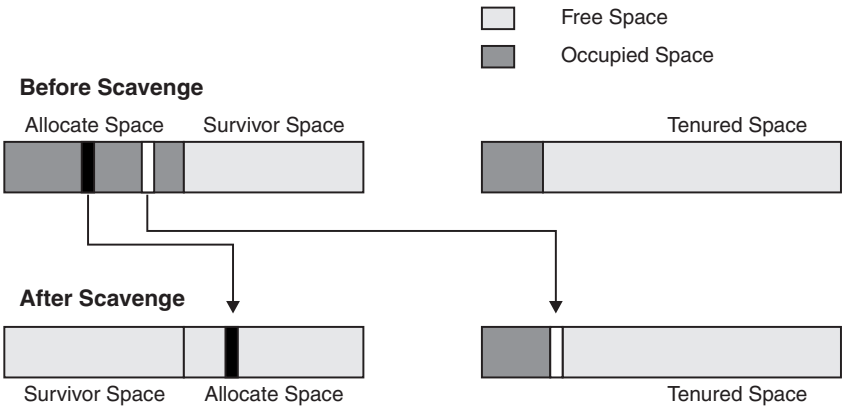
The Generational Concurrent Garbage Collector has been introduced in Java 5.0 from IBM. A generational garbage collection strategy is well suited to an application that creates many short-lived objects, as is typical of many transactional applications.

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.

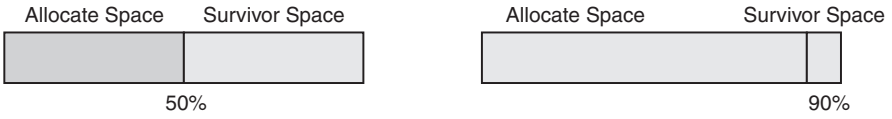
Tenure age

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, and also benefit from Non-Uniform Memory Architecture (NUMA) characteristics on modern server hardware.

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

The Java heap is split into potentially thousands of equal sized areas called "regions". Each region can be collected independently, 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.

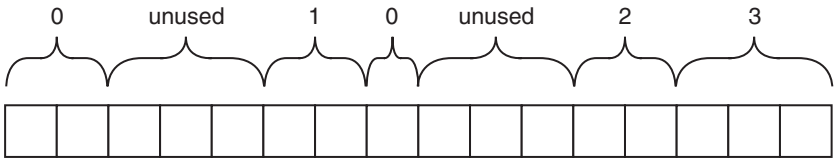
From time to time, the collector starts a Global Mark Phase (GMP) to look for more opportunities to reclaim memory. Because PGC operations see only subsets of the heap during each collection, abandoned objects might remain in the heap. This issue is like the "floating garbage" problem seen by concurrent 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.

NUMA awareness

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

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

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

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

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

Partial Garbage Collection

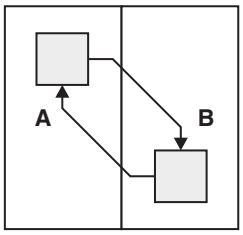
A Partial Garbage Collection (PGC) reclaims memory by using either a Copy-Forward or Mark-Compact operation on the Java heap.

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:



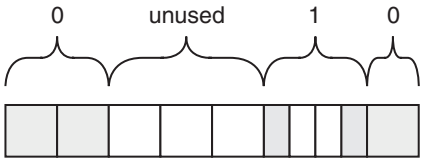
If only A or B is collected, one half of the cycle keeps the other alive. However, a GMP can see that these objects are unreachable.

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.

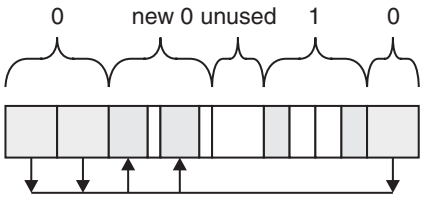
Copy-Forward operation

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

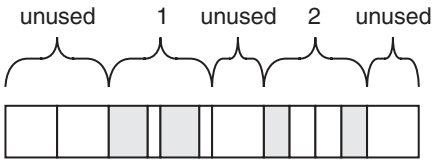
This diagram shows the Java heap before the Copy-Forward operation:



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



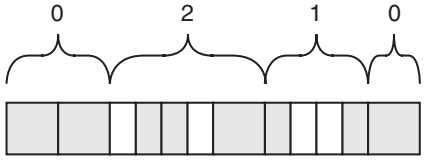
This diagram shows the Java heap after the Copy-Forward operation, where region ages have been incremented:



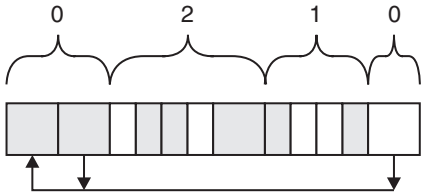
Mark-Compact operation

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

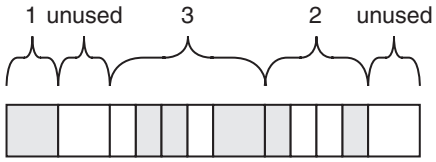
This diagram shows the Java heap before the Mark-Compact operation:



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

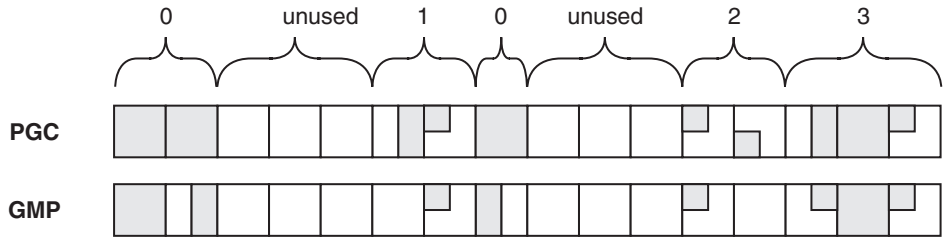
A Global Mark Phase (GMP) takes place on the entire Java heap, finding, and marking abandoned objects for garbage collection.

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.

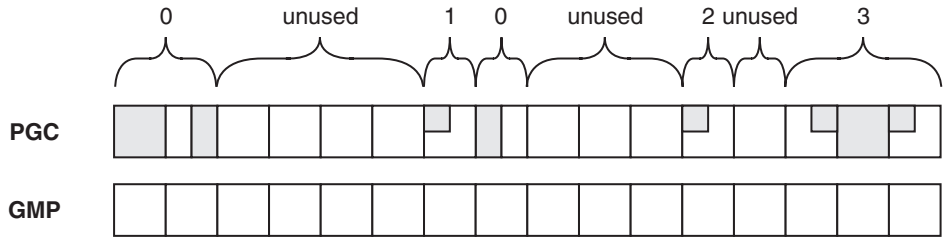
GMP also performs some work concurrently with the application threads. This concurrent mark operation is based purely on background threads, which allows idle processors to complete work, no matter how quickly the application is allocating memory. This concurrent mark operation is unlike the concurrent 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 30.

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 multi-threaded and runs on hardware that demonstrates NUMA characteristics

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

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

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

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

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

unload unused classes and class loaders on every partial collect. This approach reduces the time these classes and class loaders remain in memory.

When not to use the policy

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

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

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

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 35), 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 diagnostics" on page 375. Switch 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 490), 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 35 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,” 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 47). This difference has significant consequences for finalizers (list item 7), which are described more fully in "Finalizers."

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 below 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 47. 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 47.

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 on top of 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 47, 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 47. 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 47. 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 490.

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

`optthruput` disables concurrent mark. If you do not have pause time

problems (indicated by erratic application response times), you can expect to get the best throughput with this option.

optavgpause enables concurrent mark. If you have problems with erratic application response times in garbage collection, you can alleviate them at the cost of some throughput when running with this option.

gencon requests the combined use of concurrent and generational GC to help minimize the time that is spent in any garbage collection pause.

balanced 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. If you have problems with application pause times, caused by global garbage collections, particularly compactions, the balanced policy might improve application performance. If you are using large systems that have Non-Uniform Memory Architecture (NUMA) characteristics, the balanced policy might further improve application throughput.

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 474 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 runtime 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 459.

For diagnosing problems with shared classes, see “Shared classes diagnostics” on page 385.

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.
- 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 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 364.

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 471.

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 compliler

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 set the appropriate command-line parameter to disable it. (See “Disabling the JIT or AOT compiler” on page 364.)

`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 has been 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 loaded by that class loader are discarded.

Can I control the JIT compilation?
Yes. See "JIT and AOT problem determination" on page 364. In addition, advanced diagnostics are available to IBM engineers.

Can I dynamically control the JIT compiler?
No. You can pass options to the JIT compiler to modify its 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 runtime.

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 is grown dynamically to accommodate the needs of the program. Space previously occupied by discarded or recompiled methods is reclaimed and reused. When the size of the code cache reaches a predefined upper limit, it stops growing. The JIT compiler will stop all future attempts to compile 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.

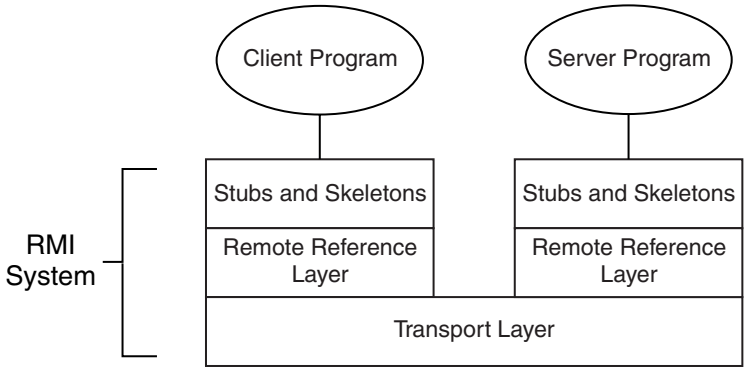
RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, supporting object-oriented polymorphism. The RMI registry is a lookup service for ports.

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:

- "Using the ORB" on page 71
- "How the ORB works" on page 75
- "Additional features of the ORB" on page 82
- "CORBA"
- "RMI and RMI-IIOP" on page 65
- "Java IDL or RMI-IIOP?" on page 65
- "RMI-IIOP limitations" on page 65
- "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 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 Version 7 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`		
`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 right 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 upper 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 square 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.

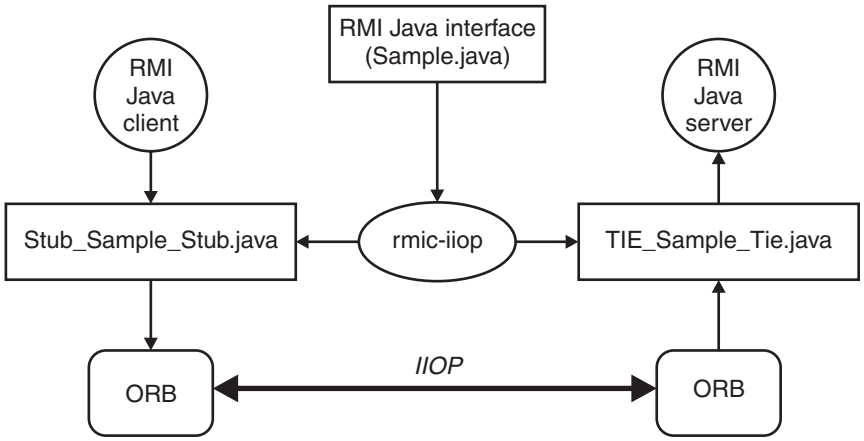
The subjects discussed here are:

- “Stub creation”
- “ORB initialization” on page 76
- “Obtaining the remote object” on page 77
- “Remote method invocation” on page 78

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 three initial bytes counting from left to right 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 246 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:

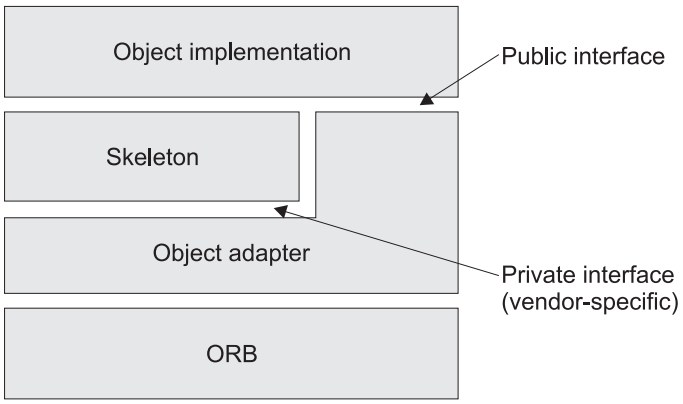
- "Portable object adapter"
- "Fragmentation" on page 84
- "Portable interceptors" on page 84
- "Interoperable Naming Service (INS)" on page 87

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:

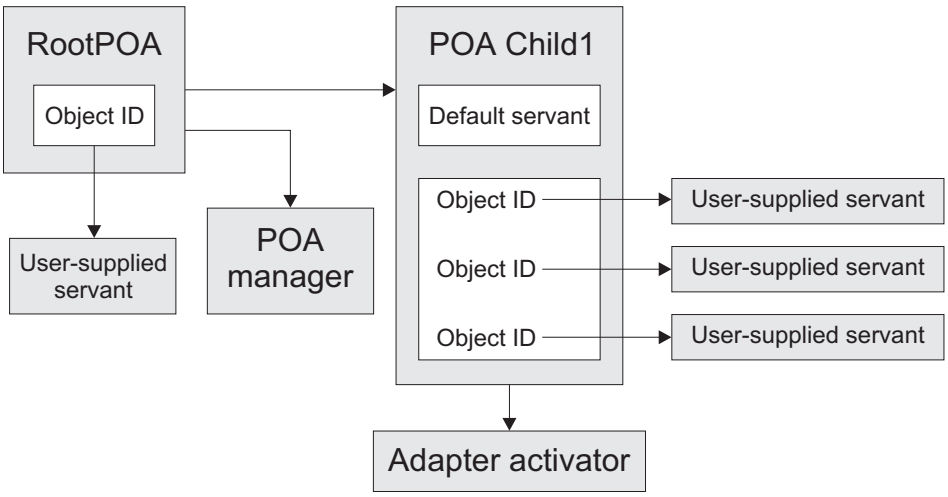
- Allow programmers to construct object and server implementations that are portable between different ORB products.
- Provide support for persistent objects. The support enables objects to persist across several server lifetimes.
- Support transparent activation of objects.
- Associate policy information 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”
- “The JNI and the Garbage Collector” on page 89
- “Copying and pinning” on page 94
- “Handling exceptions” on page 96
- “Synchronization” on page 96
- “Debugging the JNI” on page 97
- “JNI checklist” on page 98

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 runtime.

   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.

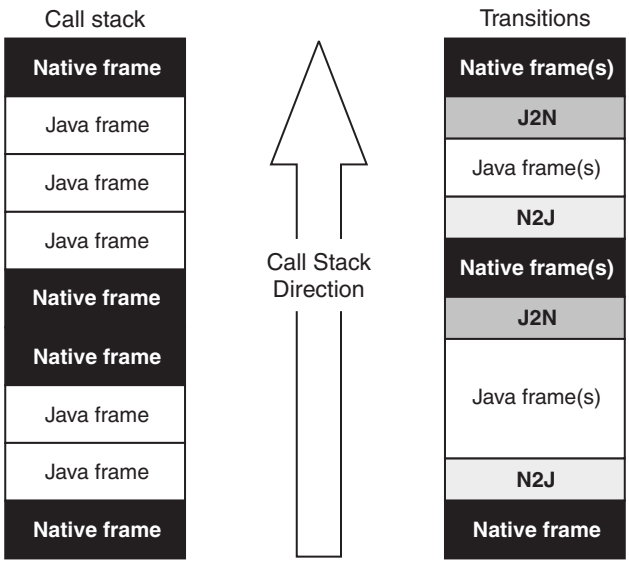
2. A valid JNI local reference in native code must be obtained:
   a. As a parameter to the native code
   b. As the return value from calling a JNI function
3. A valid JNI global reference must be obtained from another valid JNI reference (global or local) by calling NewGlobalRef or NewWeakGlobalRef.
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:

- 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 runtime. 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.

Remember	Outcome of nonadherence
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 95.	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 other IBM JVMs

Considerations when migrating from earlier versions of the IBM SDK and JRE.

The IBM SDK for AIX, v7 contains many new features and functions compared with earlier releases. For an overview, read the "What's new" on page 6 section.

If you are migrating from an older IBM Runtime Environment, note that:

- The default garbage collection policy in force is now the Generational Concurrent garbage collector, known as Gencon. For an overview, see "Generational Concurrent Garbage Collector" on page 37.
- For additional industry compatibility information, see "Version compatibility."
- For additional deprecated API information, see Oracle's Java 7 Deprecated API List: http://download.oracle.com/javase/7/docs/api/deprecated-list.html

Version compatibility

In general, any applet or application that ran with a previous version of the SDK should run correctly with the IBM SDK for AIX, v7. Classes 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/overview/index.html

Supported environments

The IBM SDK for AIX, v7 is supported on certain hardware platforms and operating systems, and is tested on specific virtualization environments.

IBM SDK for AIX

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

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

Note: IBM POWER 3 is no longer supported.

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

© Copyright IBM Corp. 2011

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

The output for a supported platform reads:

```
Model Architecture: chrp
```

The following minimum operating system levels are supported:

Table 1. AIX environments tested

Operating system	32-bit SDK	64-bit SDK
AIX 6.1 TL5	Yes	Yes
AIX 7.1.0.0	Yes	Yes

Virtualization software

For information about the virtualization software tested, see "Support for virtualization software" on page 493.

Additional information for AIX

Important information for IBM SDK and Runtime Environment for AIX, Java Technology Edition, Version 7.

AIX APARs required for IBM SDK for AIX, v7.

To avoid problems when using Java, ensure that you have these AIX APARs installed:

Table 2. AIX APARs required for IBM SDK for AIX, v7

AIX version	APAR number
6.1.5.0	IZ73931
6.1.5.4	IZ83856
6.1.6.0	IZ81170
6.1.6.15	IZ81962
7.1.0.15	IZ86109

Environment variables

The environment variable **LDR_CNTRL=MAXDATA** is not supported for 64-bit processes. Use **LDR_CNTRL=MAXDATA** only on 32-bit processes.

Graphics terminal

If you are using IBM SDK and Runtime Environment for AIX, Java Technology Edition, Version 7 on 64-bit, with UTF-8 locale and the local graphics terminal uses the UTF-8 locale, you might see an exception from `java.io.Console`.

On AIX 6.1, the exception is:

```
IZ97736: CANNOT CONTROL TTY ATTRIBUTE BY USING 64BIT PROGRAM
```

For more information, see the APAR https://www-304.ibm.com/support/docview.wss?uid=isg1IZ97736.

On AIX 7.1, the exception is:

```
IZ97912: CANNOT CONTROL TTY ATTRIBUTE BY USING 64BIT PROGRAM
```

For more information, see the APAR https://www-304.ibm.com/support/docview.wss?uid=isg1IZ97912.

Use of non-UTF8 CJK locales

If you are using one of the supported non-UTF8 CJK locales, you must install one of these file sets.

```
X11.fnt.ucs.ttf (for ja_JP or Ja_JP)
X11.fnt.ucs.ttf_CN (for zh_CN or Zh_CN)
X11.fnt.ucs.ttf_KR (for ko_KR)
X11.fnt.ucs.ttf_TW (for zh_TW or Zh_TW)
```

Note: The installation images are available on the AIX base CDs. Updates are available from the AIX fix distribution website.

When using the `zh_TW.IBM-eucTW` locale on 64-bit AIX 6.1, you might get a result that uses `ISO-8859-1` instead of `IBM-eucTW`, in response to the following command:

```
$ LANG=zh_TW locale charmap
```

The different return might affect operation of the IBM SDK for AIX, v7. If you encounter this effect, contact IBM Support for more information.

Java 2D graphics

If you want to use the improved Java 2D graphics pipeline, based on the X11 XRender extension, you must install the `libXrender.so` library, version 0.9.3 or later.

Chapter 4. Installing and configuring the SDK and Runtime Environment

You can install the IBM Java SDK and Runtime Environment from an InstallAnywhere package, or from installp packages.

InstallAnywhere packages

The InstallAnywhere packages are archive packages. Use these packages when you want to install the product files without any configuration. Fore more information, see "Installing from an InstallAnywhere package" on page 106. Packages are provided for both the JRE and SDK.

installp packages

Use these packages when you want to install the product with associated configuration, such as the setting of environment variables. Only SDK packages are provided.

This package is required:

- Java7.sdk (license, base SDK, Web Start, and dt.jar)
- Java7_64.sdk (license, base SDK and dt.jar)

These packages are optional:

- Java7.samples (demos)
- Java7.source (src.jar)
- Java7.msg.*$LANG* (Localized messages)
- Java7_64.samples (demos)
- Java7_64.source (src.jar)
- Java7_64.msg.*$LANG* (Localized messages)

$LANG is one of the following locales. These packages do not include any files but pull in required Unicode TrueType fonts, if not already installed, for these locales:

- Zh_CN
- zh_CN
- ko_KR
- Ja_JP
- ja_JP
- Zh_TW
- zh_TW

The SDK is installed in the directory:

`/usr/java7[_64]/`

The following user-configurable files are installed to `/etc/java7[_64]/` to support a configuration where the files are not shared:

- `jre/lib/jaxp.properties`
- `jre/lib/logging.properties`

© Copyright IBM Corp. 2011

- `jre/lib/management/jmxremote.access`
- `jre/lib/management/jmxremote.password.template`
- `jre/lib/management/management.properties`
- `jre/lib/management/snmp.acl`
- `jre/lib/management/snmp.acl.template`
- `jre/lib/security/java.policy`
- `jre/lib/security/java.security`
- `jre/lib/security/javaws.policy`
- `jre/lib/xalan.properties`
- `jre/lib/xerces.properties`

There are symbolic links in `/usr/java7[_64]/` pointing to the files in `/etc/java7[_64]/`.

Installing from an InstallAnywhere package

These packages provide an interactive program that guides you through the installation options. You can run the program as a graphical user interface, or from a system console.

About this task

The InstallAnywhere packages have a `.bin` file extension.

Procedure

- To install the package in an interactive way, complete an attended installation.
- To install the package without any additional user interaction, complete an unattended installation. You might choose this option if you want to install many systems.

Results

The product is installed.

Note: Do not interrupt the installation process, for example by pressing Ctrl+C. If you interrupt the process, you might have to reinstall the product. For more information, see "Interrupted installation" on page 108.

Completing an attended installation

Install the product from an InstallAnywhere package, in an interactive way.

Before you begin

Check the following conditions before you begin the installation process:

- You must have a user ID with root authority.

Procedure

1. Download the installation package file to a temporary directory.
2. Change to the temporary directory.
3. Start the installation process by typing `./package.bin` at a shell prompt, where *package* is the name of the package that you are installing.

4. Select a language from the list shown in the installer window, then click **Next**. The list of available languages is based on the locale setting for your system.
5. Read the license agreement, using the scroll bar to reach the end of the license text. To proceed with the installation you must accept the terms of the license agreement. To accept the terms, select the radio button, then click **OK**.

   Note: You cannot select the radio button to accept the license agreement until you have read to the end of the license text.
6. You are asked to choose the target directory for the installation. If you do not want to install into the default directory, click **Choose** to select an alternative directory, by using the browser window. When you have chosen the installation directory, click **Next** to continue.
7. You are asked to review the choices that you made. To change your selection, click **Previous**. If your choices are correct, click **Install** to proceed with installation.
8. When the installation process is complete, click **Done** to finish.

Completing an unattended installation

If you have more than one system to install, and you already know the installation options that you want to use, you might want to use the unattended installation process. You install once by using the attended installation process, then use the resulting response file to complete further installations without any additional user interaction.

Procedure

1. Create a response file by completing an attended installation. Use one of the following options:
   - Use the GUI and specify that the installation program creates a response file. The response file is called `installer.properties`, and is created in the installation directory.
   - Use the command line and append the -r option to the attended installation command, specifying the full path to the response file. For example:

     ```
     ./package -r /path/installer.properties
     ```

   Example response file contents:

   ```
   INSTALLER_UI=silent
   USER_INSTALL_DIR=/my_directory
   ```

   In this example, `/my_directory` is the target installation directory that you chose for the IBM SDK or IBM JRE.
2. Optional: If required, edit the response file to change options.

   Note: The packages have the following known issue: installations that use a response file use the default directory even if you change the directory in the response file. If a previous installation exists in the default directory, it is overwritten.

   If you are creating more than one response file, each with different installation options, specify a unique name for each response file, in the format `myfile.properties`.
3. Optional: Generate a log file. Because you are installing silently, no status messages are displayed at the end of the installation process. To generate a log file that contains the status of the installation, complete the following steps:
   a. Set the required system properties by using the following command.

```
export _JAVA_OPTIONS="-Dlax.debug.level=3 -Dlax.debug.all=true"
```

b. Set the following environment variable to send the log output to the console.

```
export LAX_DEBUG=1
```

4. Start an unattended installation by running the package installer with the **-i** silent option, and the **-f** option to specify the response file. For example:

```
./package -i silent -f /path/installer.properties 1>console.txt 2>&1
./package -i silent -f /path/myfile.properties 1>console.txt 2>&1
```

You can use a fully qualified path or relative path to the properties file. In these examples, the string `1>console.txt 2>&1` redirects installation process information from the stderr and stdout streams to the `console.txt` log file in the current directory. Review this log file if you think there was a problem with the installation.

Note: If your installation directory contains multiple response files, the default response file, `installer.properties` is used.

Interrupted installation

If the package installer is unexpectedly stopped during installation, for example if you press Ctrl+C, the installation is corrupted and you cannot uninstall or reinstall the product. If you try to uninstall or reinstall you might see the message `Fatal Application Error`.

About this task

To solve this problem, delete files and reinstall, as described in the following steps.

Procedure

1. Delete the `\var\.com.zerog.registry.xml` registry file.
2. Delete the directory containing the IBM SDK or IBM JRE installation, if it was created. For example `/usr/java7`.
3. Run the installation program again.

Known issues and limitations

The InstallAnywhere packages have some known issues and limitations.

- The installation package GUI does not support the Orca screen-reading program. You can use the unattended installation mode as an alternative to the GUI.
- If you install the package, then attempt to install again in a different mode, for example console or silent, you might see the following error message:

```
Invocation of this Java Application has caused an InvocationTargetException.
This application will now exit
```

  You should not see this message if you installed by using the GUI mode and are running the installation program again in console mode.
- If you change the installation directory in a response file, and then run an unattended installation by using that response file, the installation program ignores the new installation directory and uses the default directory instead. If a previous installation exists in the default directory, it is overwritten.

Relocating an installp package

If you install the product from an InstallAnywhere package, you can choose the installation directory during the installation process. If you install from the installp packages, the product is installed in /usr/java7[_64]/ by default. To install the product in another directory, use the AIX relocation commands.

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

Commands

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

installp_r
Install the SDK:

```
installp_r -a -Y -R /<Install Path>/ -d '.' Java7_64.sdk
```

Remove the SDK:

```
installp_r -u  -R /<Install Path>/ Java7_64.sdk
```

lsusil List the user-defined installation paths.

```
lsusil
```

lslpp_r
Find details of installed products.

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

rmusil Remove existing user-defined installation paths.

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

Upgrading the SDK

If you are upgrading the SDK from a previous release, back up all the configuration files and security policy files before you start the upgrade.

What to do next

After the upgrade, you might have to restore or reconfigure these files because they might have been overwritten during the upgrade process. Check the syntax of the new files before restoring the original files because the format or options for the files might have changed.

Verification

Before you begin

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

```
unset LIBPATH
unset CLASSPATH
unset JAVA_COMPILER
unset JAVA_HOME
export PATH=/usr/java7[_64]/jre/bin:/usr/java7[_64]/bin:$PATH
```

Remember: **JAVA_HOME** is used in V1.1.x releases. It is not required to set **JAVA_HOME** in this release.

About this task

When you issue the command:

```
java -version
```

you see output like the following messages: For AIX 32-bit:

```
java version "1.7.0"
Java(TM) SE Runtime Environment (build pap3270-20110319_02)
IBM J9 VM (build 2.6, JRE 1.7.0 IBM J9 2.6 AIX ppc-32 jvmap3270-20110318_55394
(JIT enabled, AOT enabled)
J9VM - R26_head_20110317_1658_B55372
JIT - dev_20110317_15164
GC - R26_head_20110317_1633_B55370
J9CL - 20110318_55394)
JCL - 20110313_01 based on Oracle 7b143
```

For AIX 64-bit:

```
java version "1.7.0"
Java(TM) SE Runtime Environment (build pap6470-20110319_02)
IBM J9 VM (build 2.6, JRE 1.7.0 IBM J9 2.6 AIX ppc64-64 jvmap6470-20110318_55394
(JIT enabled, AOT enabled)
J9VM - R26_head_20110317_1658_B55372
JIT - dev_20110317_15164
GC - R26_head_20110317_1633_B55370
J9CL - 20110318_55394)
JCL - 20110313_01 based on Oracle 7b143
```

where dates, times, and specific build numbers might be different.

What to do next

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

If the directory `.hotjava` does not exist, the applet viewer creates the directory `.hotjava` in your home directory. To confirm that the directory has been created, use the command:

```
ls -a ~
```

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 AIX to find programs and utilities, such as **javac**, **java**, and **javadoc** tool, from any current directory. To display the current value of your **PATH**, type the following at a command prompt:

```
echo $PATH
```

To add the Java launchers to your path:

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

   ```
   export PATH=/usr/java7[_64]/bin:
           /usr/java7[_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.

Chapter 5. Developing Java applications

The SDK for AIX 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 117 section for more details about what is supported by XL XP-J 1.1.

XL TXE-J 1.0

© Copyright IBM Corp. 2011

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 /etc/java7[_64]/jre/lib/stax.properties.
   - **For other factories.** The value of the service provider in the file /etc/java7[_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 using a lower split limit.

To set the split limit, use the **-SPLITLIMIT** option when using the **Process** or **Compile** commands, or the **http://www.ibm.com/xmlns/prod/xltxe-j/split-limit** transformer factory attribute when using the transformer factory. The split limit can be between 100 and 2000. When setting the split limit manually, use the highest split limit possible for best performance.

4. XL TXE-J might need more memory than the XSLT4J compiler. If you are running out of memory or performance seems slow, increase the size of the heap using the **-Xmx** option.
5. Migrate your application to use the new attribute keys. The old transformer factory attribute keys are deprecated. The old names are accepted with a warning.

Table 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 runtime, compile the stylesheet as part of your build process and put the compiled classes on your classpath. Use the org.apache.xalan.xsltc.cmdline.Compile command to compile the stylesheet and set the **http://www.ibm.com/xmlns/prod/xltxe-j/use-classpath** transformer factory attribute to `true` to load the classes from the classpath.
   - If your application will use the same stylesheet during multiple runs, set the **http://www.ibm.com/xmlns/prod/xltxe-j/auto-translet** transformer factory attribute to `true` to automatically save the compiled stylesheet to disk for reuse. The compiler will use a compiled stylesheet if it is available, and compile the stylesheet if it is not available or is out-of-date. Use the **http://www.ibm.com/xmlns/prod/xltxe-j/destination-directory** transformer

factory attribute to set the directory used to store compiled stylesheets. By default, compiled stylesheets are stored in the same directory as the stylesheet.

- If your application is a long-running application that reuses the same stylesheet, use the transformer factory to compile the stylesheet and create a Templates object. You can use the Templates object to create Transformer objects without recompiling the stylesheet. The Transformer objects can also be reused but are not thread-safe.
- 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.

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

Property name	Supported?
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

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)
http://javax.xml.transform.stax.StAXResult/feature feature	Yes	No	Yes
http://javax.xml.transform.sax.SAXTransformerFactory/feature feature	Yes	Yes	Yes
http://javax.xml.transform.sax.SAXTransformerFactory/feature/xmlfilter feature	Yes	Yes	Yes
http://javax.xml.XMLConstants/feature/secure-processing feature	Yes	Yes	Yes
http://xml.apache.org/xalan/features/incremental attribute	Yes	No	No
http://xml.apache.org/xalan/features/optimize attribute	Yes	No	No
http://xml.apache.org/xalan/properties/source-location attribute	Yes	No	No
translet-name attribute	N/A	Yes	Yes (with new name)
destination-directory attribute	N/A	Yes	Yes (with new name)
package-name attribute	N/A	Yes	Yes (with new name)
jar-name attribute	N/A	Yes	Yes (with new name)
generate-translet attribute	N/A	Yes	Yes (with new name)
auto-translet attribute	N/A	Yes	Yes (with new name)
use-classpath attribute	N/A	Yes	Yes (with new name)
enable-inlining attribute	No	Yes	No (obsolete in TL TXE-J)
indent-number attribute	No	Yes	Yes (with new name)
debug attribute	No	Yes	Yes (with new name)
Java extensions	Yes	Yes (abbreviated syntax only, xalan:component/xalan:script constructs not supported)	
JavaScript extensions	Yes	No	No
Extension elements	Yes	No	No
EXSLT extension functions	Yes	Yes (excluding dynamic)	Yes (excluding dynamic)
redirect extension	Yes	Yes (excluding redirect:open and redirect:close)	Yes

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)
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 /etc/java7[_64]/jre/lib.
- Uncomment the entries in the `jaxp.properties` file in /etc/java7[_64]/jre/lib.
- Copy the `jaxp.properties.sample` file to `jaxp.properties` in /etc/java7[_64]/jre/lib. Uncomment the entries in the `jaxp.properties` file. Create a symbolic link to the `jaxp.properties` file from the /usr/java7[_64]/jre/lib directory.
- 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.

The SDK includes a Plug-in for the AIX debugger DBX. Although the DBX Plug-in is supplied as part of the SDK, it is not supported. However, IBM will accept bug reports.

More information about problem diagnosis using Java can be found in Chapter 9, “Troubleshooting and support,” on page 211.

Java Debugger (JDB)

The Java Debugger (JDB) is included in the SDK for AIX. 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 32-bit or 64-bit JVM

Some Java applications must be able to determine whether they are running on a 32-bit JVM or on a 64-bit JVM. For example, if your application has a native code library, the library must be compiled separately in 32- and 64-bit forms for platforms that support both 32- and 64-bit modes of operation. In this case, your application must load the correct library at runtime, because it is not possible to mix 32- and 64-bit code.

About this task

The system property **com.ibm.vm.bitmode** allows applications to determine the mode in which your JVM is running. It returns the following values:

- 32 - the JVM is running in 32-bit mode
- 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. Performs the necessary JVM cleanup

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. 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 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

Table 5. Signals used by the JVM (continued)

Signal Name	Signal type	Description	Disabled by -Xrs	Disabled by -Xrs:sync
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
No Name (40)	Control	An AIX reserved signal. Used by the AIX JVM for internal control purposes.	Yes	No
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
SIGRTMIN (50)	Control	Used by the JVM for internal control purposes.	No	No
SIGRTMAX (2)	Control	Used by the SDK.	No	No
SIGCHLD (17)	Control	Used by the SDK for internal control.	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.

Do not use the **-qflttrap** C compiler setting because it provides the possibility of SIGTRAPs being generated, which might then affect the JIT. If you want to have floating point exceptions generated, include this call in your code so that it generates a SIGFPE signal:

```
fp_trap( FP_TRAP_SYNC)
```

If you install a signal handler for signal numbers 5 (SIGTRAP) or 58 (SIGRECONFIG), you affect JVM performance because these signals are used for internal control purposes.

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

The libjsig.a 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_RESTART - This is always set.

The libjsig.a 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,/usr/java7[_64]/.

To use libjsig.a:

- Link it with the application that creates or embeds a JVM:

  ```
  cc_r -q64 <other compile/link parameter> -L/usr/java7[_64]/jre/bin
  -ljsig -L/usr/java7[_64]/jre/bin/j9vm -ljvm java_application.c
  ```

  Note: Use `xlc_r` or `xlC_r` in place of `cc_r` if that is how you usually call the compiler or linker.

Writing JNI applications

Valid Java Native Interface (JNI) version numbers that programs can specify on the JNI_CreateJavaVM() API call are: JNI_VERSION_1_2(0x00010002) and JNI_VERSION_1_4(0x00010004).

Restriction: Version 1.1 of the JNI is not supported.

This version number determines only the level of the JNI to use. The actual level of the JVM that is created is specified by the JSE libraries (that is, v7). 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 32- and the other for 64-bit, use the **com.ibm.vm.bitmode** system property to determine if you are running with a 32- or 64-bit JVM and choose the appropriate library.

Supported compilers

These compilers have been tested with the IBM SDK.

The IBM XL C/C++ compiler V10.1 is supported for AIX 32-bit on PowerPC®.

The IBM XL C/C++ compiler V10.1 is supported for AIX 64-bit on PowerPC.

JNI compatibility

There is a readme file and example programs in the /usr/java7[_64]/demo/jni directory. The demos can be optionally installed with the Java6_64.samples package.

If you are writing a C or C++ program that uses the JNI Invocation API (that is, the program creates a Java Virtual Machine and calls Java code), you might want to ensure that the following variables are set appropriately. By default, all the Java launchers that are shipped with the SDK (for example, **java**, **jar**) set up these environment variables to the values that are specified as follows:

```
export AIXTHREAD_SCOPE=S
export AIXTHREAD_MUTEX_DEBUG=OFF
export AIXTHREAD_RWLOCK_DEBUG=OFF
export AIXTHREAD_COND_DEBUG=OFF
```

When you build a C or C++ program that uses the invocation API, your LIBPATH must include the directories containing the JVM's shared libraries, `/usr/java7[_64]/jre/lib/ppc[64]/` and `/usr/java7[_64]/jre/lib/ppc[64]/j9vm`, as well as the directories that contain the application's shared libraries.

You must build:

- 32-bit executables and shared objects with the **-qarch=ppc** option
- JNI executables and shared objects as a 32-bit executable or shared object
- 64-bit executables and shared objects with the **-qarch=ppc** option
- JNI executables and shared objects for the 64-bit SDK as a 64-bit program or shared object. Use the **-q64** option.

Runtime linking is supported and applications can be linked with the **-brtl** loader option.

You are recommended to compile your native methods (C or C++ functions called by Java) into AIX shared objects (dynamically loaded libraries). For example, if your native methods are stored in the file nm.c, you could create the shared object with the following command:

```
cc_r -qmkshrobj [-qarch=ppc | -q64] -I /usr/java7[_64]/include
-o libnm.a nm.c
```

The `-qmkshrobj` option disables runtime linking. For more details about shared object files, runtime linking, and the use of `cc` and `ld` command-line options, see:

- *Developing and Porting C and C++ Applications on AIX* at http://www.redbooks.ibm.com/abstracts/sg245674.html
- The C and C++ compiler documentation Web site at http://www.ibm.com/software/awdtools/xlcpp/library
- The AIX online documentation at http://www16.boulder.ibm.com/pseries/en_US/infocenter/base/aix.htm

Before running a Java program that uses native methods, ensure that **LIBPATH** contains the list of directories that are holding the native methods' shared objects. For more information about building AIX shared objects, see *C and C++ Application Development on AIX*. Go to http://www.ibm.com/redbooks and search for "SG245674".

If you set the setuid or setgid attribute on JNI native code programs, that setting changes the effective **LIBPATH** environment variable. This change might cause unexpected or incorrect behavior with those programs. For more details about this usage, see *Developing and Porting C and C++ Applications on AIX* at http://www.redbooks.ibm.com/abstracts/sg245674.html, section 2.3.3.

When building a C or C++ program that uses the JNI Invocation API to create a Java virtual machine and calls Java code, use the:

- **-L** option to add /usr/lib and /lib to the list of directories that are searched for shared objects. All programs need shared objects stored in these directories.
- **-L** option to add `/usr/java7[_64]/jre/lib/ppc[64]/` and `/usr/java7[_64]/jre/lib/ppc[64]/j9vm` to the list of directories that are searched for shared objects. These directories contain the Java SDK shared libraries. You will also want to link with libjvm.so (using the **-ljvm** option).

For example, this command builds a C program (invAPITest.c) that uses the JNI Invocation API:

```
cc_r [-qarch=ppc | -q64] -I/usr/java7[_64]/include
-o invAPITest
-L/usr/lib
-L/lib
-L/usr/java7[_64]/jre/lib/ppc[64]/j9vm
-L/usr/java7[_64]/jre/lib/ppc[64]/
-ljvm invAPITest.c
```

When executing a C or C++ program that uses the JNI Invocation API to run Java classes, ensure that the class path is set up correctly to enable the Java Virtual Machine (JVM) to find your class files. If you modify Java's boot class path, include the SDK files that are necessary to run your applications.

To determine whether a C or C++ program that is using the JNI Invocation API was built with the **-bM:UR** option, use the command:

```
>dump -ov <program name>
                        ***Object Module Header***
# Sections    Symbol Ptr    # Symbols     Opt Hdr Len   Flags
         4    0x00000d4e          155              72   0x1002
Flags=( EXEC DYNLOAD )
Timestamp = "Sep 11 13:09:13 2002"
Magic = 0x1df  (32-bit XCOFF)
                        ***Optional Header***
Tsize        Dsize       Bsize       Tstart      Dstart
0x000004b8  0x000001a8  0x00000004  0x10000128  0x200005e0

SNloader     SNentry     SNtext      SNtoc       SNdata
0x0004      0x0002      0x0001      0x0002      0x0002

TXTalign     DATAalign   TOC         vstamp      entry
0x0002      0x0003      0x20000724  0x0001      0x20000704

maxSTACK     maxDATA     SNbss       magic       modtype
0x00000000  0x00000000  0x0003      0x010b        UR
```

```
>dump -X64 -ov <program name>
                        ***Object Module Header***
# Sections      Symbol Ptr      # Symbols       Opt Hdr Len     Flags
         4      0x0001a728           1305               120     0x1002
Flags=( EXEC DYNLOAD DEP_SYSTEM )
Timestamp = "14 Oct 03:26:43 2005"
Magic = 0x1f7  (64-bit XCOFF)

                        ***Optional Header***
Tsize        Dsize       Bsize       Tstart      Dstart
0x000127f8  0x00001b80  0x00000470  0x1000001f8  0x1100009f0

SNloader     SNentry     SNtext      SNtoc       SNdata
0x0004      0x0002      0x0001      0x0002      0x0002

TXTalign     DATAalign   TOC         vstamp      entry
0x0005      0x0003      0x110002158  0x0001      0x110002040

maxSTACK     maxDATA     SNbss       magic       modtype
0x00000000  0x00000000  0x0003      0x010b        UR
```

If the **modtype** is not **UR**, you can use the **LDR_CNTRL** environment variable to make programs behave as though they were compiled with the `-bM:UR` binder option. For example:

```
export LDR_CNTRL=USERREGS
```

If you need to specify multiple options with **LDR_CNTRL**, separate those options with the "@" symbol.

Java threads that are created by the SDK use the POSIX pthreads model that is supported on AIX. Currently, this is on a 1-to-1 mapping with the kernel threads. When developing a JNI program, you must run with a 1-to-1 thread model and system contention scope if creating pthreads in your own program. You can control this by using the following environment setting:

```
export AIXTHREAD_SCOPE=S
```

Another option is to preset the thread's scope attribute to **PTHREAD_SCOPE_SYSTEM** using the AIX pthread_attr_setscope function when the thread is created. For more details on thread model and system contention scope, see:

http://publibn.boulder.ibm.com/doc_link/en_US/a_doc_lib/aixprggd/genprogc/understanding_threads.htm#D3A446E559manu

http://publibn.boulder.ibm.com/doc_link/en_US/a_doc_lib/aixprggd/genprogc/threads_sched.htm#D3A44A4324manu

From Java V5.0, you can store native methods in these ways:

Shared object
A shared object is a single object file that has the SRE (Shared REusable) bit set in the XCOFF header. The SRE bit tells the linker that this file is linked dynamically. These files typically have a name of the form <*filename*>.o, but they can also be named lib<*name*>.a to allow the linker to search for them with the **-lname** option; but these are not archive library files.

Shared library
A shared library is an "ar" format archive library in which one or more of the archive members is a shared object. Note that this library can also contain non-shared object files that are statically linked. A shared library has the name in the form lib<*name*>.a. This form allows the linker to search for libraries with the **-lname** option.

Programs can also link dynamically to AIX shared libraries and shared objects using the dlopen() family of subroutines. The SDK links in this way when it loads native libraries (for example, System.load(), System.loadLibrary(), Runtime.getRuntime().load(), Runtime.getRuntime().loadLibrary()).

For information about dlopen, see:

dlopen Subroutine

For information about AIX loading and linking mechanisms, see: AIX Linking and Loading Mechanisms

To load an AIX shared library, make a call to:

```
System.loadLibrary("<library>(<member>)")
```

where <*library*> is the name of the shared library archive and <*member*> is the name of an archive member. For example:

```
System.loadLibrary("libShared.a(libSample.o)")
```

Example of using AIX shared libraries

This example takes you through the process of using native shared libraries with a Java application on AIX.

Procedure

1. Create a sample application, `Sample.java`.

```
public class Sample
{
   public native void printFromNative( );

   public static void main( String[] args )
   {
      Sample sample = new Sample( );
      sample.printFromNative( );
   }
```

```
    static
    {
       String sharedLibrary = "libShared.a(libSample.o)";
       try
       {
          System.loadLibrary( sharedLibrary );
       }
       catch ( Exception e )
       {
          System.out.println( "ERROR: Unable to load " + sharedLibrary );
          e.printStackTrace( );
       }
    }
}
```

2. Compile `Sample.java`.

   ```
   javac Sample.java
   ```

3. Use **javah** to create a header file for the native code.

   ```
   javah Sample
   ```

4. Create a file called `Sample.c`.

   ```
   #include <stdio.h>
   #include "Sample.h"

   JNIEXPORT void JNICALL Java_Sample_printFromNative( JNIEnv * env, jobject obj )
   {
      printf( "Printing from native\n" );
   }
   ```

5. Compile `Sample.c` into `libSample.o`.

   ```
   cc_r -bM:SRE -bnoentry -bexpall -I/usr/java7[_64]/include Sample.c
   -o libSample.o -q64
   ```

6. Create an archive shared library `libShared.a`.

   ```
   ar -X32 -v -q libShared.a libSample.o
   ar -X64 -v -q libShared.a libSample.o
   ```

7. Run the Sample class.

   ```
   LIBPATH=. java Sample
   ```

   The program will output:

   ```
   Printing from native
   ```

Results

You should now be able to use the same framework to access native shared libraries from Java applications.

Support for thread-level recovery of blocked connectors

Four new IBM-specific SDK classes have been added to the com.ibm.jvm package to support the thread-level recovery of Blocked connectors. The new classes are packaged in `core.jar`.

These classes allow you to unblock threads that have become blocked on networking or synchronization calls. If an application does not use these classes, it must end the whole process, rather than interrupting an individual blocked thread.

The classes are:

public interface InterruptibleContext
Defines two methods, isBlocked() and unblock(). The other three classes implement InterruptibleContext.

public class InterruptibleLockContext
A utility class for interrupting synchronization calls.

public class InterruptibleIOContext
A utility class for interrupting network calls.

public class InterruptibleThread
A utility class that extends java.lang.Thread, to allow wrapping of interruptible methods. It uses instances of InterruptibleLockContext and InterruptibleIOContext to perform the required isBlocked() and unblock() methods depending on whether a synchronization or networking operation is blocking the thread.

Both InterruptibleLockContext and InterruptibleIOContext work by referencing the current thread. Therefore if you do not use InterruptibleThread, you must provide your own class that extends java.lang.Thread, to use these new classes.

The Javadoc information for these classes is provided with the SDK in the `docs/content/apidoc` directory.

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 Version 7. 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.iop.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

Table 6. Methods affected when running with Java SecurityManager (continued)

Class/Interface	Method	Required permission
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 v7 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 141), 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."
4. Change all the places in your code where there is a Java cast of a remote interface to use javax.rmi.PortableRemoteObject.narrow().
5. Do not depend on distributed garbage collection (DGC) or use any of the RMI DGC facilities. Use PortableRemoteObject.unexportObject() to make the ORB release its references to an exported object that is no longer in use.
6. Regenerate the RMI stubs and ties using the **rmic** command with the **-iiop** option. This will produce stub and tie files with the following names:

   _<*implementationName*>_Tie.class
   _<*interfaceName*>_Stub.class
7. Before starting the server, start the CosNaming server (in its own process) using the `tnameserv` command The CosNaming server uses the default port number of 2809. If you want to use a different port number, use the **-ORBInitialPort** parameter.
8. When starting client and server applications, you must specify some system properties. When running an application, you can specify properties on the command line:

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
     -Djava.naming.provider.url=iiop://<hostname>:2809
     <appl_class>
```

9. If the client is an applet, you must specify some properties in the applet tag. For example:

```
java.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
java.naming.provider.url=iiop://<hostname>:2809
```

   This example uses the default name service port number of 2809. If you specify a different port in the previous step, you need to use the same port number in the provider URL here. The <*hostname*> in the provider URL is the host name that was used to start the CosNaming server.

Results

Your application can now communicate with CORBA objects using RMI-IIOP.

Connecting IIOP stubs to the ORB:

When your application uses IIOP stubs, as opposed to JRMP stubs, you must properly connect the IIOP stubs with the ORB before starting operations on the

IIOP stubs (this is not necessary with JRMP stubs). This section discusses the extra 'connect' step required for the IIOP stub case.

The PortableRemoteObject.exportObject() call only creates a Tie object and caches it for future usage. The created tie does not have a delegate or an ORB associated. This is known as explicit invocation.

The PortableRemoteObject.exportObject() happens automatically when the servant instance is created. The servant instance is created when a PortableRemoteObject constructor is called as a base class. This is known as implicit invocation.

Later, when the application calls PortableRemoteObject.toStub(), the ORB creates the corresponding Stub object and associates it with the cached Tie object. But because the Tie is not connected and does not have a delegate, the newly created Stub also does not have a delegate or ORB.

The delegate is set for the stub only when the application calls Stub.connect(orb). Thus, any operations on the stub made before the ORB connection is made will fail.

The Java Language to OMG IDL Mapping Specification (http://www.omg.org/cgi-bin/doc?formal/01-06-07) says this about the Stub.connect() method:

> "The connect method makes the stub ready for remote communication using the specified ORB object orb. Connection normally happens implicitly when the stub is received or sent as an argument on a remote method call, but it is sometimes useful to do this by making an explicit call (e.g., following deserialization). If the stub is already connected to orb (has a delegate set for orb), then connect takes no action. If the stub is connected to some other ORB, then a RemoteException is thrown. Otherwise, a delegate is created for this stub and the ORB object orb."

For servants that are not POA-activated, Stub.connect(orb) is necessary as a required setup.

Restrictions when running RMI programs over IIOP:

A list of limitations when running RMI programs over IIOP.

To make existing RMI programs run over IIOP, observe the following restrictions.

- Make sure all constant definitions in remote interfaces are of primitive types or String and evaluated at compile time.
- Do not use Java names that conflict with IDL mangled names generated by the Java-to-IDL mapping rules. See section 28.3.2 of the Java Language to OMG IDL Mapping Specification for more information: http://www.omg.org/cgi-bin/doc?formal/01-06-07
- Do not inherit the same method name into a remote interface more than once from different base remote interfaces.
- Be careful when using names that are identical other than their case. The use of a type name and a variable of that type with a name that differs from the type name in case only is supported. Most other combinations of names that are identical other than their case are not supported.
- Do not depend on runtime sharing of object references to be preserved exactly when transmitting object references to IIOP. Runtime sharing of other objects is preserved correctly.

- Do not use the following features of RMI, which do not work in RMI-IIOP:
  - RMISocketFactory
  - UnicastRemoteObject
  - Unreferenced
  - The Distributed Garbage Collector (DGC) interfaces

Additional information

Information about thread safety, working with other ORBs, the difference between UnicastRemoteObject and PortableRemoteObject, and known limitations.

Servers must be thread safe

Because remote method invocations on the same remote object might execute concurrently, a remote object implementation must be thread-safe.

Interoperating with other ORBs

RMI-IIOP should interoperate with other ORBs that support the CORBA 2.3 specification. It will not interoperate with older ORBs, because older ORBs cannot handle the IIOP encodings for Objects By Value. This support is needed to send RMI value classes (including strings) over IIOP.

Note: Although ORBs written in different languages should be able to interoperate, the Java ORB has not been fully tested with other vendors' ORBs.

When do I use UnicastRemoteObject vs PortableRemoteObject?

Use UnicastRemoteObject as the superclass for the object implementation in RMI programming. Use PortableRemoteObject in RMI-IIOP programming. If PortableRemoteObject is used, you can switch the transport protocol to either JRMP or IIOP during runtime.

Known limitations

- JNDI 1.1 does not support *java.naming.factory.initial*=com.sun.jndi.cosnaming.CNCtxFactory as an Applet parameter. Instead, it must be explicitly passed as a property to the InitialContext constructor. This capability is supported in JNDI 1.2.
- When running the Naming Service on Unix based platforms, you must use a port number greater than 1024. The default port is 2809, so this should not be a problem.

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 top of your `.java` file from: `import com.ibm.math.*;` to `import java.math.*;`.

AIX native threads

On AIX, programs can set the priority of system contention scope threads. Calls to java.lang.Thread.setPriority() will change the priority of Java threads running on AIX.

For more information about AIX dynamic thread scheduling, see:

http://publibn.boulder.ibm.com/doc_link/en_US/a_doc_lib/aixprggd/genprogc/threads_sched.htm

For more information about AIX 6.1 thread scheduling, see:

pthread_setschedparam Subroutine and Scheduling Threads.

JNDI

The IBM SDK for AIX, v7 provides a unified interface, the Java Naming and Directory Interface (JNDI), to the naming and directory services.

These naming and directory services are supported by JNDI:

- Lightweight Directory Access Protocol (LDAP)
- Corba Object Services (COS) Naming Service
- RMI Registry
- File system
- Domain Name Service (DNS)

HPROF Performance Profiler and JVMPI

The Java Virtual Machine Profiling Interface (JVMPI) has been extended to include profiling in the IBM JIT. These additional definitions are listed in `jvmpi.h`.

GetCurrentThreadCpuTime

To enable the JVMPI function GetCurrentThreadCPUTime, you must enable AIX resource collection by setting the environment variable **AIXTHREAD_ENRUSG**.

About this task

When the resource collection is enabled, the HPROF profiling agent, started with **-Xrunhprof:cpu=times** provides meaningful data in the HPROF output.

Procedure

Type the following command at the shell prompt:

```
export AIXTHREAD_ENRUSG=ON
```

Results

You will get the following message when performance profiling is switched on:

```
Warning: Performance profiling is enabled and can cause
performance degradation.
Warning: You can unset AIXTHREAD_ENRUSG to disable
performance profiling.
```

What to do next

Make sure that the resource collection is disabled after your profiling session because the added processor usage when tracking CPU time for individual threads affects performance. Resource collection should never be enabled in a production environment. To disable the resource collection, do one of the following:

```
unset AIXTHREAD_ENRUSG
export AIXTHREAD_ENRUSG=OFF
```

By default, resource collection is disabled.

Known restriction

The JVMPI_EVENT_INSTRUCTION_START event is not currently supported in the IBM SDK for AIX, v7. It might be added in a future release.

Support for XToolkit

The IBM SDK for AIX, v7 includes XToolkit by default. You need XToolkit when using the SWT_AWT bridge in Eclipse to build an application that uses both SWT and Swing.

Restriction: Motif is no longer supported and will be removed in a later release.

Related links:

- An example of integrating Swing into Eclipse RCPs: http://eclipsezone.com/eclipse/forums/t45697.html
- Reference Information in the Eclipse information center: http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/awt/SWT_AWT.html
- Set up information is available on the 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.

The Attach API is enabled by default.

Security considerations

Security for the Java Attach API is handled by UNIX user and group file permissions.

The Java Attach API creates files and directories in a common directory. The common directory, subdirectories, and files in it, have UNIX file permissions. To prevent "spoofing" attacks, change the ownership of the common directory to ROOT or another privileged user ID.

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 can attach other applications to the target process.
- For Java 7, access to the files or directories owned by a process is controlled by user permissions only; group access (read/write) is disabled.
- The common directory uses the sticky bit to prevent a user from deleting or replacing a subdirectory belonging to another user. To preserve the security of this mechanism, set the ownership of the common directory to ROOT.
- The subdirectory for a process is accessible only by members of the same UNIX group as the owner of a process. For Java 7, read/write access is restricted to the owner only, except for the `attachNotificationSync` file. The `attachNotificationSync` file can be written to by all users. 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 only by the owner. Information about the target process can be read only by the owner or a member of the same group as the owner. For Java 7, read/write access is restricted to the owner only.

You must secure access to the Java Attach API capability to ensure that only authorized users or processes can connect to another virtual machine. If you do not intend to use the Java Attach API capability, disable this feature using 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 called `.com_ibm_tools_attach`, which is created in the system temporary directory. The system property **java.io.tmpdir** holds the value of the system temporary directory. On non-Windows systems, the system temporary directory is typically `/tmp`. To modify the working directory, use the Java system property **com.ibm.tools.attach.directory**; for example:

```
-Dcom.ibm.tools.attach.directory=/working
```

The 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
```

An unsuccessful attempt to start the Attach API results in one of the following exceptions:

- `com.ibm.tools.attach.AgentLoadException`
- `com.ibm.tools.attach.AgentInitializationException`
- `com.ibm.tools.attach.AgentNotSupportedException`
- `java.io.IOException`

A useful reference for information about the Attach API can be found at http://download.oracle.com/javase/7/docs/technotes/guides/attach/index.html.

The IBM implementation of the Attach API corresponds approximately to the Oracle Corporation implementation. However, if your application originally used com.sun.tools.attach.* methods or classes, you must modify and recompile the application to use the com.ibm.tools.attach.* implementation.

Plug-in, Applet Viewer and Web Start

The Java plug-in is used to run Java applications in the browser. The **appletviewer** is used to test applications designed to be run in a browser. Java Web Start is used to deploy desktop Java applications over a network, and provides a mechanism for keeping them up-to-date.

Using the Java plug-in

The Java plug-in is a web browser plug-in. You use the Java plug-in to run applets in the browser.

Allow enough time for applets to finish loading, otherwise your browser might seem to "stop". For example, if you click **Back** and then click **Forward** while an applet is loading, the HTML pages might be unable to load.

The Java plug-in is documented at: http://download.oracle.com/javase/7/docs/technotes/guides/plugin/developer_guide/contents.html.

Supported browsers

The Java plug-in supports the following browsers: Mozilla and Mozilla Firefox.

Table 7. Browsers supported by the Java plug-in on 32-bit AIX

Browser	Supported Versions
Mozilla	1.7.13, Available at http://www.ibm.com/servers/aix/browsers/ and the Mozilla for AIX CD.

Table 8. Browsers supported by the Java plug-in on 64-bit AIX

Browser	Supported Versions
Firefox	1.7, 2.0, 3.5, 3.6. Available at http://www.ibm.com/servers/aix/browsers/.

Installing the Java plug-in

To install the Java plug-in, symbolically link it to the plug-in directory for your browser.

The Java plug-in is based on the Mozilla Open JVM Integration initiative, which is used with most Mozilla products and derivatives, including Firefox.

You must symbolically link the plug-in, rather than copy it, so that the browser and plug-in can locate the JVM.

Follow these steps to make the Java plug-in available to Mozilla Firefox users:

1. Log in as root user.
2. Change to the Firefox plug-in directory.
   ```
   cd /usr/local/mozilla-firefox/plugins/
   ```
   Note: The directory name is different for different Linux distributions.
3. Create a symbolic link to the plug-in. For Firefox 1.7, use the command:

```
ln -s /opt/ibm/java-<arch>-60/jre/plugin/<arch>/ns7/libjavaplugin_oji.so .
```

Where `<arch>` is the architecture of your system. For Firefox 3.x, use the command:

```
ln -s /opt/ibm/java-<arch>-60/jre/lib/<arch>/libnpjp2.so .
```

Changing the properties of the Java Plug-in

You can change the properties of the Java Plug-in from the control panel, which can be run as a stand-alone Java application.

About this task

To start this Java application, run the script:

```
/usr/java7[_64]/jre/bin/ControlPanel
```

Common Document Object Model (DOM) support

Because of limitations in particular browsers, you might not be able to implement all the functions of the org.w3c.dom.html package.

One of the following errors is thrown:

- sun.plugin.dom.exception.InvalidStateException
- sun.plugin.dom.exception.NotSupportedException

Using DBCS parameters

The Java plug-in supports double-byte characters (for example, Chinese Traditional BIG-5, Korean, and Japanese) as parameters for the tags <APPLET>, <OBJECT>, and <EMBED>. You must select the correct character encoding for your HTML document so that the Java plug-in can parse the parameter.

About this task

Specify character encoding for your HTML document by using the `<META>` tag in the `<HEAD>` section like this:

```
<meta http-equiv="Content-Type" content="text/html; charset=big5">
```

This example tells the browser to use the Chinese BIG-5 character encoding to parse the HTML file.

Working with applets

With the Applet Viewer, you can run one or more applets that are called by reference in a Web page (HTML file) by using the <APPLET> tag. The Applet Viewer finds the <APPLET> tags in the HTML file and runs the applets, in separate windows, as specified by the tags.

Because the Applet Viewer is for viewing applets, it cannot display a whole Web page that contains many HTML tags. It parses only the <APPLET> tags and no other HTML on the Web page.

Running and debugging applets with the Applet Viewer

Use the following commands to run and debug an applet with the Applet Viewer.

Procedure

- To run an applet with the Applet Viewer, enter the following command: `appletviewer <name>`.

<*name*> is one of the following:

– The file name of an HTML file that calls an applet.

– The URL of a Web page that calls an applet.

For example, to start the Applet Viewer on an HTML file that calls an applet, type at a shell prompt:

```
appletviewer $HOME/<filename>.html
```

Where *filename* is the name of the HTML file.

To start the Applet Viewer on a Web page, type at a shell prompt:

```
appletviewer http://java.sun.com/applets/jdk/1.4/demo/applets/NervousText/
example1.html
```

The Applet Viewer does not recognize the **charset** option of the <

```
META
```

> tag. If the file that the Applet Viewer loads is not encoded as the system default, an I/O exception might occur. To avoid the exception, use the **-encoding** option when you run **appletviewer**. For example:

```
appletviewer -encoding JISAutoDetect sample.html
```

- To debug an applet with the Applet Viewer, use the `debug` parameter with the `appletviewer` command.

  For example:

  ```
  cd demo/applets/TicTacToe
  ../../../bin/appletviewer -debug example1.html
  ```

  You can find documentation about how to debug applets using the Applet Viewer at the Oracle Web site: .http://download.oracle.com/javase/7/docs/technotes/guides/plugin/developer_guide/debugger.html

Java Applet Viewer and the classpath

If you use the Applet Viewer to run an applet that is in the **CLASSPATH**, you might get an AccessControlException in Swing. Because the **CLASSPATH** implicitly contains the current directory ".", this exception might occur if you run the Java Plug-in in the same directory that the applet class itself is in.

To work around this problem, ensure that:

- No **CLASSPATH** references exist to the applet that you are attempting to run in the Java Plug-in or the **appletviewer**.
- You are not running the applet from the same directory that the class is in.

Using Web Start

Java Web Start is used for Java application deployment.

With Web Start, you can start and manage applications directly from the Web. Applications are cached to minimize installation times. Applications are automatically upgraded when new versions become available.

Web Start supports these command-line arguments documented at http://download.oracle.com/javase/7/docs/technotes/guides/javaws/developersguide/syntax.html#resources:

- -verbose
- -version
- -showversion
- -help

- -X
- -ea
- -enableassertions
- -da
- -disableassertions
- -esa
- -enablesystemassertions
- -dsa
- -disablesystemassertions
- -Xint
- -Xnoclassgc
- -Xdebug
- -Xfuture
- -Xrs
- -Xms
- -Xmx
- -Xss

Web Start also supports **-Xgcpolicy** to set the garbage collection policy.

The Autodownload option in the Java Control Panel is set to **Always** by default. This option enables a user without administration privileges to download the JRE from the location specified in the JNLP file.

For more information about Web Start, see:

- http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136112.html and
- http://download.oracle.com/javase/7/docs/technotes/guides/javaws/index.html.

For more information about deploying applications, see:

- http://download.oracle.com/javase/7/docs/technotes/guides/deployment/index.html.

Running Web Start

Web Start can be run from a Web page or the command line. Web Start applications are stored in the Java Application Cache.

About this task

You can start Web Start in a number of different ways.

Procedure

- Select a link on a Web page that refers to a `.jnlp` file. If your browser does not have the correct association to run Web Start applications, select the `/usr/java7[_64]/jre/bin/javaws` command from the **Open/Save** window to start the Web Start application.
- At a shell prompt, type:

  ```
  javaws <URL>
  ```

  Where <*URL*> is the location of a `.jnlp` file.

- If you have used Java Web Start to open the application in the past, use the Java Application Cache Viewer. At a shell prompt, type:

  ```
  /usr/java7[_64]/jre/bin/javaws -viewer
  ```

  All Java Web Start applications are stored in the Java Application Cache. An application is downloaded only if the latest version is not in the cache.

WebStart Secure Static Versioning

Static versioning allows Web Start applications to request a specific JVM version on which those applications will run. Because static versioning also allows applications to exploit old security vulnerabilities on systems that have been upgraded to a new JVM, Secure Static Versioning (SSV) is now used by default.

With SSV, the user is warned before running any unsigned Web Start application that requests a specific JVM, if the requested JVM is installed. Signed applications and applications that request the latest version of the JVM run as usual.

You can disable SSV by setting the **deployment.javaws.ssv.enabled** property in the `deployment.properties` file to false.

Distributing Java applications

Java applications typically consist of class, resource, and data files.

When you distribute a Java application, your software package probably consists of the following parts:
- Your own class, resource, and data files
- AIX Runtime Environment (optional)
- An installation procedure or program

To run your application, a user needs the Runtime Environment for AIX. The SDK for AIX software contains a Runtime Environment. However, you cannot assume that your users have the SDK for AIX software installed.

Your application can either make the SDK for AIX a prerequisite or include a version of the SDK that is specifically for the purpose of redistribution. The SDK for AIX license does not allow you to redistribute any of the SDK files installed in /usr/java7[_64]/ by **installp**. You can redistribute the SDK files in the j732redist.tar, j764redist.tar or j732redist.tar.gz, j764redist.tar.gz files (after viewing and agreeing to the associated online license) available from the AIX Java Web site: http://www.ibm.com/developerworks/java/jdk/aix/. Click the **Download and service information** link near the bottom of the page and follow the links to the Java download page.

Java Communications API (JavaComm)

The Java Communications (API) package (JavaComm) is an optional package provided for use with the Runtime Environment for AIX. You install JavaComm independently of the SDK or Runtime Environment.

The JavaComm API gives Java applications a platform-independent way of performing serial and parallel port communications for technologies such as voice mail, fax, and smartcards.

The Java Communications API supports Electronic Industries Association (EIA)-232 (RS232) serial ports and Institute of Electrical and Electronics Engineers (IEEE) 1284 parallel ports and is supported on systems with the IBM Version 7 Runtime Environment.

Using the Java Communications API, you can:

- List ports on a system
- Open and claim ownership of ports
- Resolve port ownership contention among applications that use Java Communications API
- Perform asynchronous and synchronous I/O port-monitoring using event notification
- Receive bean-style events describing state changes on the port

The API is shipped in the Java5_64.ext.commapi fileset (optionally installable).

Location of the Java Communications API files

By default, the Java Communications API files are installed in the `/usr/java7[_64]/` directory.

The files and their structure are:

- `jar/comm.jar`
- `jar/commtest.jar`
- `jar/tools/BlackBox.jar`
- `jar/tools/ParallelBlackBox.jar`
- `lib/javax.comm.properties`
- `lib/libLinuxSerialParallel.so`

Mapping serial ports

The Java communications API supports an unlimited number of tty's, with ttyn mapped to COM(n+1).

For example:

```
/dev/tty0 -> COM1
/dev/tty1 -> COM2
...
/dev/tty9 -> COM10
/dev/tty10 -> COM11
...
```

If a tty is not available, the corresponding COM port is not available. For example, if tty0 and tty2 are available but tty1 is not, COM1 and COM3 are available but COM2 is not.

Configuring the Java Communications API

To use the Java Communications API, you must change the access mode of serial and parallel ports, and set the **PATH** if you did not set it when you installed Java.

About this task

See “Setting the path” on page 110.

Specifying devices in the `javax.comm.properties` file

Use the javax.comm.properties file to specify the devices and drivers that are available to the Java Communications API and whether they are parallel or serial. Do not change this file without a very clear understanding of its use.

About this task

The following properties must be defined:

```
driver=<driver_name>
serpath0=<serial_port_device>
parpath0=<parallel_port_device>
```

For example:

```
# Implementation specific driver
driver=com.sun.comm.LinuxDriver

# Paths to server-side serial port devices
serpath0 = /dev/ttyS0
serpath1 = /dev/ttyS1

# Paths to server-side parallel port devices
parpath0 = /dev/parport0
parpath1 = /dev/parport1
```

For parallel port access, use `/dev/parport<n>`. `/dev/lp<n>` is not supported in Javacomm version 3.0.

Printing limitation with the Java Communications API

When printing with the Java Communications API, you might have to select “Form feed”, “Continue”, or a similar option on the printer.

The Java Communications API documentation

You can find API documentation and samples for the Java Communications API at the Oracle Web site.

http://www.oracle.com/technetwork/java/index-jsp-141752.html.

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**, except that **javaw** has no associated console window. Use **javaw** when you do not want a command prompt window to be displayed. The **javaw** launcher displays a window with error information if it fails.

Usage

The JVM searches for the initial class (and other classes that are used) in three sets of locations: the bootstrap class path, the installed extensions, and the user class path. The arguments that you specify after the class name or jar file name are passed to the main function.

The **java** and **javaw** commands have the following syntax:

```
java [ options ] <class> [ arguments ... ]
java [ options ] -jar <file.jar> [ arguments ... ]
javaw [ options ] <class> [ arguments ... ]
javaw [ options ] -jar <file.jar> [ arguments ... ]
```

Parameters

[*options***]**
: Command-line options to be passed to the runtime environment.

<class>
: Startup class. The class must contain a main() method.

<file.jar>
: Name of the jar file to start. It is used only with the **-jar** option. The named jar file must contain class and resource files for the application, with the startup class indicated by the Main-Class manifest header.

[*arguments ...***]**
: Command-line arguments to be passed to the main() function of the startup class.

Obtaining version information

You obtain the IBM build and version number for your Java installation using the **-version** option.

© Copyright IBM Corp. 2011

Procedure

1. Open a shell prompt.
2. Type the following command:

   ```
   java -version
   ```

   You will see information similar to:Exact build dates and versions will change.

What to do next

You can also list the version information for all available `jar` files on the class path, the boot class path, and in the extensions directory. Type the following command:

```
java -Xjarversion
```

You will see information similar to:

```
...
/usr/java7[_64]/jre/lib/ext/ibmpkcs11impl.jar  VERSION: 1.0 build_20070125
/usr/java7[_64]/jre/lib/ext/dtfjview.jar
/usr/java7[_64]/jre/lib/ext/xmlencfw.jar  VERSION: 1.00, 20061011  LEVEL: -20061011

...
```

The information available varies for each `jar` file and is taken from the **`Implementation-Version`** and **`Build-Level`** properties in the manifest of the `jar` file.

Specifying Java options and system properties

You can specify Java options and system properties on the command line, by using an options file, or by using an environment variable.

About this task

These methods of specifying Java options are listed in order of precedence. Rightmost options on the command line have precedence over leftmost options; for example, if you specify:

```
java -Xint -Xjit myClass
```

The **`-Xjit`** option takes precedence.

Procedure

1. By specifying the option or property on the command line. For example:

   ```
   java -Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump MyJavaClass
   ```

2. By creating a file that contains the options, and specifying it on the command line using **`-Xoptionsfile=`**<*file*>.
3. By creating an environment variable called **`IBM_JAVA_OPTIONS`** containing the options. For example:

   ```
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait
   -Xdisablejavadump"
   ```

Standard options

The definitions for the standard options.

See “JVM command-line options” on page 459 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 474 for more information) to control the output. For additional information, see "Verbose garbage collection logging" on page 375.

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.

Working with the `LIBPATH` environment variable

The **LIBPATH** environment variable tells AIX applications, such as the JVM, where to find shared libraries when they are located in a different directory from the directory that is specified in the header section of the program.

For example, the header section of the **java** command is as follows:

```
> dump -X64 -H /usr/java7[_64]/jre/bin/java
/usr/java7[_64]/jre/bin/java:

                      ***Loader Section***
                    Loader Header Information
VERSION#         #SYMtableENT     #RELOCent        LENidSTR
0x00000001       0x0000003f       0x0000006d       0x00000090

#IMPfilID        OFFidSTR         LENstrTBL        OFFstrTBL
0x00000006       0x00000b24       0x00000099       0x00000bb4


                      ***Import File Strings***
INDEX  PATH                          BASE            MEMBER
0     /usr/lib:/lib

1                                    libc.a          shr.o
2                                    libC.a          shr.o
3                                    libpthreads.a   shr_comm.o
4                                    libpthreads.a   shr_xpg5.o
5                                    libbsd.a        shr.o
```

Index 0 in the example contains the list of directories that are searched for shared objects if **LIBPATH** is not specified. If **LIBPATH** is set, the specified directories are searched for shared objects before the directories listed in Index 0 of the header.

The shared libraries for the SDK are in `/usr/java7[_64]/jre/lib/ppc[64]/` and `/usr/java7[_64]/jre/lib/ppc[64]/j9vm`. The SDK launcher programs, including **java**, **javac**, and **jar** automatically search these directories. If Java is installed as an AIX file set, the parent directory is /usr/java7[_64]/, but packages that bundle Java might use different directories. This path is already set by the Java launcher programs such as **java**, **javac**, or **jar**.

Set the **LIBPATH** if either of the following conditions applies:

- You are using other shared libraries (including JNI native libraries you use or develop). Set the **LIBPATH** to include the directory or directories that contain your libraries.
- You are using the JNI Invocation API to call Java code from your C/C++ application. Set the **LIBPATH** to include the directories that contain the JVM libraries in addition to the directories that contain your own libraries.

Working with the LDR_CNTRL environment variable

The **LDR_CNTRL** environment variable controls the way AIX handles the memory space available to a 32-bit program and the page sizes used in each segment. The POWER4 and later PowerPC processors support the use of 16 MB large pages in addition to the default 4 KB pages. The POWER5+ and later PowerPC processors add support for two new page sizes, 64 KB and 16 GB.

Controlling the available memory space

The SDK typically sets appropriate **LDR_CNTRL=MAXDATA** settings automatically for the size of the Java heap. However, if you explicitly set the **LDR_CNTRL=MAXDATA** environment variable the SDK does not override the value you have specified. Do not override the automatic settings unless you understand the AIX memory models. For more information about the AIX memory models and the automatic settings used by the SDK, see “SDK use of AIX large program support” on page 167.

Page sizes

AIX v6.1 and later operating systems support 16 MB, 64 KB, and 16 GB pages. The 16 MB and 16 GB pages require AIX system configuration changes. For information about using 16 MB pages with AIX, see Large pages. For information about using either 64 KB or 16 GB pages with AIX, see Multiple page size support. The default AIX page size is 4 KB.

The **LDR_CNTRL=LARGE_PAGE_DATA** environment variable can be used under the AIX v6.1 and later operating systems to control the use of 16 MB pages for the native data area and native heap of a program. You can use 16 MB large pages by setting **LDR_CNTRL=LARGE_PAGE_DATA=Y**. Using large pages might improve the performance of Java applications that require a large amount of native heap space. In particular, the native heap is used for code generated by the just-in-time compiler and also Java applications with many compiled methods. The reason is that both might benefit from using 16 MB pages for the native heap.

Use **LDR_CNTRL** variants to independently control the use of different page sizes for the text (**TEXTPSIZE**), stack (**STACKPSIZE**) and native data or heap (**DATAPSIZE**) areas. See Multiple page size application support for general information about these variants.

An example of the use of **TEXTPSIZE**, **STACKPSIZE**, and **DATAPSIZE** variants is:

```
LDR_CNTRL=TEXTPSIZE=4K@STACKPSIZE=64K@DATAPSIZE=64K
```

This example uses 4 KB pages for text, 64 KB pages for stack and 64 KB pages for the native data and native heap areas. A **DATAPSIZE** setting overrides any **LARGE_PAGE_DATA** setting.

In the default AIX 5L™ 32-bit process address space model, the initial thread stack size and data of a process are located in the same PowerPC 256 MB segment. Only one page size can be used in a segment. If different page sizes are specified for the stack and data of a standard 32-bit process, the smaller page size is used for both. For information about how to use different segments for the stack and data of a process, see: Multiple page size application support considerations.

When page sizes are set using **LDR_CNTRL**, the SDK does not automatically set **MAXDATA** and it is set to **MAXDATA=0**. To use a Java heap larger than 2.5 GB, set **MAXDATA** to a different value. See "SDK use of AIX large program support" on page 167 for more information about possible settings. For example:

```
LDR_CNTR=MAXDATA=0XB0000000@DSA@TEXTPSIZE=4K@STACKPSIZE=64K@DATAPSIZE=64K
```

The new 64 KB pages are general purpose. Most workloads see a benefit by using 64 KB pages for text, stack, native data, and the Java heap. The 16 GB pages are intended only for use in high performance environments.

Note: Use the **-Xlp** option variants to request that the JVM allocates the Java heap with a specific size of pages.

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 v7 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.

The Stack Execution Disable (SED) feature in the AIX 5300-03 Recommended Maintenance package stops code from executing in data areas (Power4 and later). For more information about this feature and how it affects the SDK, see "AIX Stack Execution Disable" on page 170.

For more information about the JIT, see "JIT and AOT problem determination" on page 364.

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 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 at a shell prompt:

  ```
  java -Djava.compiler=NONE <class>
  ```

- Use the **-Xint** option on the JVM command line. Type the following 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 at a shell prompt:

```
java -version
```

If the JIT is not in use, a message is displayed that includes the following:

```
(JIT disabled)
```

If the JIT is in use, a message is displayed that includes the following:
`(JIT enabled)`

What to do next

For more information about the JIT, see “The JIT compiler” on page 56.

Specifying 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 is left for a storage request, or by an explicit System.gc() call.

Garbage collection can significantly affect application performance, so the IBM virtual machine provides various methods of optimizing the way garbage collection is carried out, potentially reducing the effect on your application.

For more detailed information about garbage collection, see “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 and its values is:

-Xgcpolicy:gencon
(Default value.) Requests the combined use of concurrent and generational GC to help minimize the time that is spent in any garbage collection pause.

-Xgcpolicy:optthruput
Delivers high throughput to applications, but at the cost of occasional pauses.

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

-Xgcpolicy:balanced
Uses a region-based layout for the Java heap. These regions are individually managed to reduce the maximum pause time on large heaps and increase the efficiency of garbage collection. The policy also uses a different object allocation strategy that improves application throughput on large systems that have Non-Uniform Memory Architecture (NUMA) characteristics. For more information about this policy, see “Balanced Garbage Collection policy” on page 39.

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 IBM SDK for Java 64-bit stores object references as 64-bit values. The **-Xcompressedrefs** command-line option causes object references to be 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.

Use **-Xcompressedrefs** in any of these situations:

- When your Java applications does not need more than a 25 GB Java heap.
- When your application uses a lot of native memory and needs the JVM to run in a small footprint.
- When you want to enable the Balanced Garbage Collection policy, using the **-Xgcpolicy:balanced** command-line option.

See "Compressed references" on page 28 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 diagnostics. In these conditions, you are recommended either to increase the heap size by using the **-Xmx** option or to reduce the number of objects in use.

For more information, see "Garbage Collector diagnostics" on page 375.

Dynamic Logical Partitioning (DLPAR) support

System resources, for instance memory and CPUs, can be dynamically added to or removed from a logical partition (LPAR) running AIX. Java applications can take advantage of any new resources. Java applications can also respond to DLPAR events using extensions to the java.lang.management API.

If you run Java applications on a single CPU LPAR and never dynamically add a CPU to that LPAR while those Java applications are running, you can improve the performance by exporting the environment variable: `export NO_LPAR_RECONFIGURATION=1`. The results vary depending on the execution characteristics of your application. Do not export this environment variable unless all the following conditions are true:

- You are running in an LPAR
- The LPAR has 1 CPU
- The LPAR will never be dynamically reconfigured to add more CPUs while Java applications are running.

Resource changes are effective immediately, so AIX does not need to be rebooted. If an administrator decreases the number of CPUs or memory allocated to an LPAR, the performance of any running SDK application might degrade.

For more information, see http://publib.boulder.ibm.com/infocenter/pseries/v5r3/topic/com.ibm.aix.prftungd/doc/prftungd/dyn_log_part.htm and the article: *Dynamic reconfiguration: Basic building blocks for autonomic computing on IBM pSeries® servers* in the following IBM Systems Journal issue: http://www.research.ibm.com/journal/sj42-1.html.

To enable applications to respond to DLPAR events, the Java v7 SDK includes IBM-specific extensions to java.lang.management. The extensions provide a Java interface to query various LPAR-specific information, and to listen for events indicating that the logical partition of the JVM has been dynamically altered. The Javadoc for this API is provided with the SDK in the `docs/content/apidoc` directory.

The launcher option, **-Xsoftmx**, is also available with the Java 6 SDK. The **–Xmx** option specifies the maximum size (hard limit) of the Java heap. The **–Xsoftmx** option specifies a lower initial maximum heap size (a "soft" limit). You can change the value of **-Xsoftmx** at run time using the java.lang.management API. The valid range of values is between the minimum heap size (**-Xms**) and the hard limit (**-Xmx**).

For example, if the JVM is running in an LPAR with 2 GB of memory available for the heap, but the amount of memory might be changed to as low as 1 GB or as high as 8 GB during the run, a suitable set of command-line options might be:

```
–Xms1g –Xsoftmx2g –Xmx8g
```

The value of **–Xms** must be less than or equal the value of **-Xsoftmx**. If unspecified, **-Xsoftmx** defaults to the value of **-Xmx**.

Live application mobility on AIX WPAR

IBM AIX Workload Partitions (WPARs) are software-created, virtualized operating system environments in a single instance of the AIX operating system. To most applications, the workload partition appears to be a separate instance of AIX. Applications in workload partitions have a private execution environment.

Applications in workload partitions are isolated in terms of process and signal, and can be isolated in file system space. Workload partitions can have their own unique users and groups. Workload partitions have dedicated network addresses and interprocess communication (IPC) is restricted to processes running in the same workload partition.

There are two forms of workload partitions:

System WPAR
: A System WPAR presents an environment like a stand-alone AIX system. A System WPAR runs most of the system services that are found in a stand-alone system and does not share writable file systems with any other WPAR or the global system.

Application WPAR
: An Application WPAR has all the process isolation that a System WPAR provides, except that it shares file system namespace with the global system and any other Application WPAR defined in the system. Other than the application itself, a typical Application WPAR runs an additional lightweight **`init`** process in the WPAR.

You can configure either WPAR type for mobility, which allows you to move running WPAR instances between physical systems using the AIX Workload Manager.

The IBM SDK for Java supports WPAR mobility. The IBM SDK for Java can also respond to WPAR mobility events and use system environment changes in the same way as when a DLPAR is reconfigured. See "Dynamic Logical Partitioning (DLPAR) support" on page 165 for more information.

For more information about WPAR, see http://www.redbooks.ibm.com/redpieces/abstracts/SG247431.html.

Using the IPv6 extensions

The IBM SDK for AIX, v7 uses the IPv6 extensions to the TCP/IP protocol by default.

If you do not want to use IPv6 protocols, you can set the property **`java.net.preferIPv4Stack`** to force the use of IPv4 protocols.

Enhanced BiDirectional support

The IBM SDK includes enhanced BiDirectional support.

For more information, see http://www.ibm.com/developerworks/java/jdk/additional/JAVABIDI.html.

The Javadoc file for the BiDirectional package is provided with the SDK in the `docs/content/apidoc` directory.

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.

Scaling support

To increase the maximum number of threads your system can support, lower the maximum native stack size using the **-Xss<size>** option.

The default native stack size is 1024 KB512 KB. A smaller setting allows for a larger number of threads. For example:

```
java -Xss<size> <other params>
```

To increase the maximum number of file descriptors your system can support, use the **ulimit** or **chuser** commands, for example:

```
ulimit -n 3000
```

or

```
chuser nofiles=3000 <user_id>
```

Use `ulimit -a` to show the current limit.

SDK use of AIX large program support

The memory space available to a 32-bit program is broken up by AIX into 16 segments of 256 MB each. The AIX memory model being used by a program determines how those segments can be used by the program. In particular, the memory model determines how many segments are available to the native heap and how many segments are available for other purposes.

The AIX memory model is controlled by the **LDR_CNTRL=MAXDATA** environment variable. For more details on the AIX memory models, see the *Large Program Support* section of the AIX *General Programming Concepts: Writing and Debugging Programs* manual available at: http://www-05.ibm.com/e-business/linkweb/publications/servlet/pbi.wss?CTY=US&FNC=SRX&PBL=SC23-4128-08. The manual provides information about using the address-space models to accommodate programs requiring data areas that are larger than conventional segmentation can handle. The SDK uses the large program support in AIX by automatically setting the **LDR_CNTRL=MAXDATA** environment variable as appropriate for allocating the Java heap.

The SDK needs space in both the native heap (for non-object allocations such as for thread stacks or JIT buffers), and space that can be used for the Java heap. In almost all instances, the automatic settings of **LDR_CNTRL=MAXDATA** are adequate. In certain instances, a user might want to override the automatic settings and manually set **LDR_CNTRL=MAXDATA**. If an explicit setting is used, the SDK does not

attempt to change it. Suboptimal settings for a particular Java application can result in out of memory errors when you run the application. The presence of these errors indicates that you should change the automatic or manual setting in use.

Automatic LDR_CNTRL=MAXDATA values

The automatic setting of the **LDR_CNTRL=MAXDATA** environment variable value is linked to the size of the Java heap used by the SDK. To simplify the setting of **LDR_CNTRL=MAXDATA**, the SDK sets an appropriate value that is based on the maximum size of the heap.

If **LDR_CNTRL=MAXDATA** is set before you start the SDK, the SDK uses the specified value. Otherwise, the SDK uses the following algorithm to set **LDR_CNTRL=MAXDATA**:

- If the heap size is greater than 3 GB, **LDR_CNTRL=MAXDATA=0@DSA** is set.
- If the heap size is greater than 2.25 GB but no more than 3 GB, **LDR_CNTR=MAXDATA=0XB0000000@DSA** is set.
- Otherwise, **LDR_CNTR=MAXDATA=0XA0000000@DSA** is set.

Here is a picture of the memory layout with **MAXDATA=0xA0000000@DSA**, used for Java heaps up to 2.25 GB:

Table 9. The default Java virtual memory model

Segment	Contents
0	AIX kernel
1	Java program
2	Primordial Stack (main program thread stack)
3	Native Heap (malloc'ed space)
4-C	Native Heap (malloc'ed space) or Memory Mapped space (mmap/shmat)
D	Shared library code
E	Memory Mapped space (mmap/shmat)
F	Shared library data

Segments 0 and 1 have a fixed usage; segment 0 is always used for the AIX kernel and segment 1 is always used for the application program code. In this case, the application program code is generally the **java** executable.

The setting of **MAXDATA=0xA0000000@DSA** has determined the usage of the other segments as follows:

- Segment 2 is used for the stack of the application program.
- Segment 3 to segment C are available to the native heap, although initially only segment 3 is reserved for the native heap. Because the JVM and JIT allocate space from the native heap, it can expand to consume additional contiguous segments.
- The Java heap is allocated in contiguous space in segment E, or from segment C and lower segments. That is, a Java heap of 256 MB or smaller uses just segment E. A Java heap of more than 256 MB uses segments C, B, ... as necessary, up to a maximum size of 2.25 GB using all of C-4. At the maximum size of 2.25 GB, the native heap cannot expand further than segment 3.

- Segment D has been allocated by the operating system for shared library code, segment F is used for shared library data. The JVM and JIT are mostly contained in shared libraries which are loaded into these segments.

The Java heap is allocated by the JVM using the AIX mmap or shmget/shmat functions. The mmap function is used when the Java heap is allocated in normal 4 KB pages. The shmget/shmat function is used when the **-Xlp** option specifies that large pages are to be used.

It is clear from this memory layout that some Java applications can have problems when using large Java heaps. If a Java heap of 2.25 GB is used, the native heap is restricted to a single 256 MB segment. If the Java application created many threads, for example, it might consume a large fraction of the native heap for thread stacks, which can make the JVM or JIT run out of native heap space. Such a situation requires more careful consideration of what size of Java heap should be used, and might motivate the use of an explicit **MAXDATA** setting.

For a Java heap larger than 2.25 GB, a different **MAXDATA** setting is needed to free up additional segments for a contiguous area large enough for the heap. With the **MAXDATA=0xB0000000@DSA** automatic setting, the memory layout changes to:

Table 10. Memory model with MAXDATA=0xB0000000@DSA

Segment	Contents
0	AIX Kernel
1	Java program
2	Primordial stack (main program thread stack)
3	Native Heap (malloc'ed space)
4-D	Native Heap (malloc'ed space) or Memory Mapped space (mmap/shmat)
E-F	Memory Mapped space (mmap/shmat)

Segments 0 and 1 are fixed in their usage. The setting of **MAXDATA=0xB0000000@DSA** has determined the usage of the other segments as follows:

- Segment 2 is used for the stack of the application program.
- Segment 3 to segment D are available to the native heap, although initially only segment 3 is reserved for the native heap. Because the JVM and JIT allocate space from the native heap, it can expand to consume additional contiguous segments. Shared libraries must also be privately loaded into this range of segments because segments D and F are not reserved for shared libraries.
- The Java heap is allocated in contiguous space from segment F and lower segments, up to a maximum size of 3 GB using all of F-4.

Using a maximum sized 3 GB heap places restrictions on the native heap memory available to the JVM and JIT. In this case, there is the additional restriction that the shared libraries must coexist with the native heap.

For a Java heap larger than 3 GB, the only option is to use **MAXDATA=0@DSA**, resulting in a memory layout as follows:

Table 11. Memory model with MAXDATA=0@DSA

Segment	Contents
0	AIX kernel
1	Java program

Table 11. Memory model with MAXDATA=0@DSA (continued)

Segment	Contents
2	Native Heap (malloc'ed space) and Primordial Stack (main invocation stack)
3-F	Memory Mapped space (mmap/shmat)

The Java heap in this case consumes all of segments 3 through F, for a maximum size of 3.25 GB. The primordial stack, native heap, and shared libraries are all packed into segment 2. It is unlikely that many Java applications could successfully run in this configuration.

Manual LDR_CNTRL=MAXDATA values

Use the **LDR_CNTRL=MAXDATA** environment variable to enable memory layouts that are different from the automatic settings.

One interesting setting is *MAXDATA=0*:

Table 12. Memory model with MAXDATA=0

Segment	Contents
0	AIX kernel
1	Java program
2	Primordial Stack (main program thread stack) and Native Heap
3-C	Memory Mapped space (mmap/shmat)
D	Shared library code
E	Memory Mapped space (mmap/shmat)
F	Shared library data

The setting of **MAXDATA=0** has determined the usage of the segments as follows:

- Segment 2 is used for the stack of the application program, and for the native heap.
- The Java heap is allocated in contiguous space in segment 3 and higher segments. That is, a Java heap of 256 MB or smaller uses just segment 3. A Java heap of more than 256 MB uses segments 3, 4, ... as necessary, up to a maximum size of 2.5 GB using all of 3-C.
- Segment D has been allocated by the operating system for shared library code. Segment F is used for shared library data. The JVM and JIT are mostly contained in shared libraries, which are loaded into these segments.

AIX Stack Execution Disable

AIX 5300-03 implements Buffer Overflow Protection (BOP) using Stack/heap Execution Disable (SED). SED prevents buffer overflow attacks by not executing code in data areas of memory. AIX system administrators control the way SED is used. Java JIT implementations generate machine code in C heap memory; therefore, Java launchers must be exempt from SED.

You make programs exempt from SED by setting the XCOFF executable file header flag DEP_EXEMPT. All Java launchers have the appropriate bit set to exempt them from the SED feature.

Applications that use their own Java launchers and create JVM instances using JNI must be explicitly patched to exempt them from SED. Use the **sedmgr** utility and verify the change using the **dump** or **sedmgr** utility.

The syntax for using these utilities is:

```
sedmgr -c exempt <launcher>
dump -X64 -ov <launcher>
```

For more details on SED, see http://publib.boulder.ibm.com/infocenter/pseries/index.jsp.

Chapter 7. Performance

You can improve the performance of the product by sharing class data between virtual machines.

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 memory-mapped cache file on disk. Sharing reduces the overall virtual storage consumption when more than one JVM shares a cache. Sharing also reduces the startup time for a JVM after the cache has been created. The shared class cache is independent of any running JVM and persists until it is deleted.

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-up 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 183.

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 upper and lower 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 175 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 175 for more information.

© Copyright IBM Corp. 2011

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 181 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.

The cache memory is protected against accidental or deliberate corruption using memory page protection. This protection is not an absolute guarantee against corruption because the JVM must unprotect pages to write to them. The only way to guarantee that a cache cannot be modified is to open it read-only.

If a Java SecurityManager is installed, classloaders, excluding the default bootstrap, application, and extension 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 182 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 the cache file is deleted manually.

Cache utilities

All cache utilities are suboptions to the **-Xshareclasses** command. See "Class data sharing command-line options" on page 175 or use **-Xshareclasses:help** to see a list of available suboptions.

Class data sharing command-line options

Class data sharing and the cache management utilities are controlled using command-line options to the Java launcher.

For options that take a <*size*> parameter, suffix the number with "k" or "K" to indicate kilobytes, "m" or "M" to indicate megabytes, or "g" or "G" to indicate gigabytes.

-Xscmaxaot<*size*>
: Sets the maximum number of bytes in the cache that can be used for AOT data. Use this option to ensure a certain amount of cache space is available for non-AOT data. By default, the maximum limit for AOT data is the amount of free space in the cache. The value of this option should not be smaller than the value of **-Xscminaot** and must not be larger than the value of **-Xscmx**.

-Xscmaxjitdata<*x*>
: Optionally applies a maximum number of bytes in the class cache that can be used for JIT data. This option is useful if you want a certain amount of cache space guaranteed for non-JIT data. If this option is not specified, the maximum limit for JIT data is the amount of free space in the cache. The value of this option must not be smaller than the value of **-Xscminjitdata**, and must not be larger than the value of **-Xscmx**.

-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 181.)

-Xshareclasses:<*suboption*>**[,**<*suboption*>**...]**
Enables class data sharing. Can take a number of suboptions, some of which are cache utilities. Cache utilities perform the required operation on the specified cache, without starting the VM. You can combine multiple suboptions, separated by commas, but the cache utilities are mutually exclusive. When running cache utilities, the message `Could not create the Java virtual machine` is expected. Cache utilities do not create the virtual machine.

Some cache utilities can work with caches from previous Java versions or caches created by JVMs with different bit-widths. These caches are referred to as "incompatible" caches.

You can use the following suboptions with the **-Xshareclasses** option:

help
Lists all the command-line suboptions.

name=<*name*>
Connects to a cache of a given name, creating the cache if it does not already exist. Also used to indicate the cache that is to be modified by cache utilities; for example, **destroy**. Use the **listAllCaches** utility to show which named caches are currently available. If you do not specify a name, the default name "sharedcc_%u" is used. %u in the cache name inserts the current user name. You can specify "%g" in the cache name to insert the current group name.

cacheDir=<*directory*>
Sets the directory in which cache data is read and written. By default, <*directory*> is `/tmp/javasharedresources`. The user must have sufficient permissions in <*directory*>. The directory must not be on an NFS mount for persistent caches.The JVM writes persistent cache files directly into the directory specified. Persistent cache files can be safely moved and deleted from the file system. Non-persistent caches are stored in shared memory and have control files that describe the location of the memory. Control files are stored in a `javasharedresources` subdirectory of the **cacheDir** specified. Do not move or delete control files in this directory. The **listAllCaches** utility, the **destroyAll** utility, and the **expire** suboption work only in the scope of a given **cacheDir**.

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 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, permissions for both new and existing cache directories are set to 777, for compatibility with earlier Java versions.

readonly
Opens an existing cache with read-only permissions. The JVM does not create a new cache with this suboption. Opening a cache read-only prevents the JVM from making any updates to the cache. It also allows the JVM to connect to caches created by other users or groups without requiring write access. By default, this suboption is not specified.

persistent
Uses a persistent cache. The cache is created on disk, which persists beyond operating system restarts. Non-persistent and persistent caches can have the same name. You must always use the **persistent** suboption when running utilities such as **destroy** on a persistent cache. You must set the **CORE_MMAP** environment variable to `yes` when using a persistent cache.

nonpersistent (default)
Uses a non-persistent cache. The cache is created in shared memory, which is lost when the operating system shuts down. Non-persistent and persistent caches can have the same name.

groupAccess
Sets operating system permissions on a new cache to allow group access to the cache. The default is user access only.

verbose
Enables verbose output, which provides overall status on the shared class cache and more detailed error messages.

verboseAOT
Enables verbose output when compiled AOT code is being found or stored in the cache. AOT code is generated heuristically. You might not see any AOT code generated at all for a small application. You can disable AOT caching using the **noaot** suboption.

verboseIO
Gives detailed output on the cache I/O activity, listing information on classes being stored and found. Each 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 401.

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 401.

modified=<*modified context*>
Used when a JVMTI agent is installed that might modify bytecode at runtime. If you do not specify this suboption and a bytecode modification agent is installed, classes are safely shared with an extra performance cost. The <*modified context*> is a descriptor chosen by the user; for example, "myModification1". This option partitions the cache, so that only JVMs using context myModification1 can share the same classes. For instance, if you run HelloWorld with a modification context and then run it again with a different modification context, all classes are stored twice in the cache. See "Runtime bytecode modification" on page 181 for more information.

reset
Causes a cache to be destroyed and then recreated when the JVM starts up. Can be added to the end of a command line as **-Xshareclasses:reset**.

destroy (Utility option)
Destroys a cache specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. A cache can be destroyed only if all JVMs using it have shut down, and the user has sufficient permissions.

destroyAll (Utility option)
Tries to destroy all caches available using the specified **cacheDir** and **nonpersistent** suboptions. A cache can be destroyed only if all JVMs using it have shut down, and the user has sufficient permissions.

expire=<*time in minutes*>
Destroys all caches that have been unused for the time specified before loading shared classes. This option is not a utility option because it does not cause the JVM to exit.

listAllCaches (Utility option)
Lists all the compatible and incompatible caches that exist in the specified cache directory. If you do not specify **cacheDir**, the default directory is used. Summary information, such as Java version and current usage is displayed for each cache.

printStats[=<*data_types*>**] (Utility option)**
Displays summary information for the cache specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. The most useful information displayed is how full the cache is and how many classes it contains. Stale classes are classes that have been updated on the file system and which the cache has therefore marked "stale". Stale classes are not purged from the cache and can be reused.

Specify one or more data types, separated by a plus symbol (+), to additionally see more detailed information about that type of cache content. Data types include AOT data, class paths and ROMMethods.See the "printStats utility" on page 403 for more information.

printAllStats (Utility option)
Displays detailed information for the cache specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. Every class is listed in chronological order, with a reference to the location from which it was loaded. AOT code for class methods is also listed.

See the "printAllStats utility" on page 407 for more information.

mprotect=[all | default | none]
By default, the memory pages containing the cache are protected at all times, unless a specific page is being updated. This protection helps prevent accidental or deliberate corruption to the cache. The cache header is not protected by default because this protection has a small performance cost. Specifying `all` ensures that all the cache pages are protected, including the header. Specifying `none` disables the page protection.

If you use the **–Xshareclasses:nonpersistent** option, set the environment variable MPROTECT_SHM to ON before starting the JVM. If you do not set this environment variable, the **-Xshareclasses:mprotect** option is ignored, whether you specify a value for the option or accept the default value, and no page protection occurs when you use a nonpersistent cache.

noBootclasspath
Prevents storage of classes 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. See the Diagnostics Guide for more information about shared class filtering.

cacheRetransformed
Enables caching of classes that have been transformed using the JVMTI RetransformClasses function.

noaot
Disables caching of AOT code. AOT code already in the shared data cache can be loaded.

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 disabled using **-Xzero:nosharebootzip**. For further details on **-Xzero** and the options available, see "JVM command-line options" on page 459.

Creating, populating, monitoring, and deleting a cache

An overview of the life-cycle of a shared class data cache including examples of the cache management utilities.

To enable class data sharing, add **-Xshareclasses[:name=<name>]** to your application command line.

The JVM either connects to an existing cache of the given name or creates a new cache of that name. If a new cache is created, it is populated with all bootstrap and application classes being loaded until the cache becomes full. If two or more JVMs are started concurrently, they populate the cache concurrently.

To check that the cache has been created, run `java -Xshareclasses:listAllCaches`. To see how many classes and how much class data is being shared, run `java -Xshareclasses:[name=<name>],printStats`. You can run these utilities after the application JVM has terminated or in another command window.

For more feedback on cache usage while the JVM is running, use the **verbose** suboption. For example, `java -Xshareclasses:[name=<name>],verbose`.

To see classes being loaded from the cache or stored in the cache, add `-Xshareclasses:[name=<name>],verboseIO` to your application command line.

To delete the cache, run `java -Xshareclasses:[name=<name>],destroy`. You usually delete caches only if they contain many stale classes or if the cache is full and you want to create a bigger cache.

You should tune the cache size for your specific application, because the default is unlikely to be the optimum size. To determine the optimum cache size, specify a large cache, using **-Xscmx**, run the application, and then use **printStats** to determine how much class data has been stored. Add a small amount to the value shown in **printStats** for contingency. Because classes can be loaded at any time during the lifetime of the JVM, it is best to do this analysis after the application has terminated. However, a full cache does not have a negative affect on the performance or capability of any JVMs connected to it, so it is acceptable to decide on a cache size that is smaller than required.

If a cache becomes full, a message is displayed on the command line of any JVMs using the **verbose** suboption. All JVMs sharing the full cache then loads any further classes into their own process memory. Classes in a full cache can still be shared, but a full cache is read-only and cannot be updated with new classes.

Performance and memory consumption

Class data sharing is particularly useful on systems that use more than one JVM running similar code; the system benefits from reduced virtual storage consumption. It is also useful on systems that frequently start and shut down JVMs, which benefit from the improvement in startup time.

The processor and memory usage required to create and populate a new cache is minimal. The JVM startup cost in time for a single JVM is typically between 0 and 5% slower compared with a system not using class data sharing, depending on how many classes are loaded. JVM startup time improvement with a populated cache is typically between 10% and 40% faster compared with a system not using class data sharing, depending on the operating system and the number of classes loaded. Multiple JVMs running concurrently show greater overall startup time benefits.

Duplicate classes are consolidated in the shared class cache. For example, class A loaded from `myClasses.jar` and class A loaded from `myOtherClasses.jar` (with

identical content) is stored only once in the cache. The **printAllStats** utility shows multiple entries for duplicated classes, with each entry pointing to the same class.

When you run your application with class data sharing, you can use the operating system tools to see the reduction in virtual storage consumption.

Considerations and limitations of using class data sharing

Consider these factors when deploying class data sharing in a product and using class data sharing in a development environment.

Cache size limits

The maximum theoretical cache size is 2 GB. The size of cache you can specify is limited by the amount of physical memory and paging space available to the system.

The cache for sharing classes is allocated using the System V IPC Shared memory mechanism. You can change the default behavior using the **-Xshareclasses:persistent** option, so that memory mapped files are created on disk and remain when the operating system is restarted. In this case you must set the environment variable *CORE_MMAP=yes*. For further information, see the **-Xshareclasses:persistent** option in the topic "JVM command-line options" on page 459.

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.

The virtual address space available to the process is 3.25 GB, depending on your system configuration. See "SDK use of AIX large program support" on page 167.

The virtual address space available to the process is 32 TB.

JVMTI RetransformClasses() is unsupported

You cannot run RetransformClasses() on classes loaded from the shared class cache.

The JVM might throw the exception UnmodifiableClassException if you attempt to run RetransformClasses(). It does not work because class file bytes are not available for classes loaded from the shared class cache. If you must use RetransformClasses(), ensure that the classes to be transformed are not loaded from the shared class cache, or disable the shared class cache feature.

Runtime bytecode modification

Any JVM using a JVM Tool Interface (JVMTI) agent that can modify bytecode data must use the **modified=**<*modified_context*> suboption if it wants to share the modified classes with another JVM.

The modified context is a user-specified descriptor that describes the type of modification being performed. The modified context partitions the cache so that all JVMs running under the same context share a partition.

This partitioning allows JVMs that are not using modified bytecode to safely share a cache with those that are using modified bytecode. All JVMs using a given modified context must modify bytecode in a predictable, repeatable manner for each class, so that the modified classes stored in the cache have the expected

modifications when they are loaded by another JVM. Any modification must be predictable because classes loaded from the shared class cache cannot be modified again by the agent.

If a JVMTI agent is used without a modification context, classes are still safely shared by the JVM, but with a small affect on performance. Using a modification context with a JVMTI agent avoids the need for extra checks and therefore has no affect on performance. A custom ClassLoader that extends java.net.URLClassLoader and modifies bytecode at load time without using JVMTI automatically stores that modified bytecode in the cache, but the cache does not treat the bytecode as modified. Any other VM sharing that cache loads the modified classes. You can use the **modified=**<*modification_context*> suboption in the same way as with JVMTI agents to partition modified bytecode in the cache. If a custom ClassLoader needs to make unpredictable load-time modifications to classes, that ClassLoader must not attempt to use class data sharing.

See "Dealing with runtime bytecode modification" on page 393 for more detail on this topic.

Operating system limitations

You cannot share classes between 32-bit and 64-bit JVMs. Temporary disk space must be available to hold cache information. The operating system enforces cache permissions.

For operating systems that can run both 32-bit and 64-bit applications, class data sharing is not permitted between 32-bit and 64-bit JVMs. The **listAllCaches** suboption lists 32-bit or 64-bit caches, depending on the address mode of the JVM being used.

The shared class cache requires disk space to store identification information about the caches that exist on the system. This information is stored in `/tmp/javasharedresources`. If the identification information directory is deleted, the JVM cannot identify the shared classes on the system and must re-create the cache. Use the **ipcs** command to view the memory segments used by a JVM or application.

Users running a JVM must be in the same group to use a shared class cache. The operating system enforces the permissions for accessing a shared class cache. If you do not specify a cache name, the user name is appended to the default name so that multiple users on the same system create their own caches by default.

Using SharedClassPermission

If a SecurityManager is being used with class data sharing and the running application uses its own class loaders, you must grant these class loaders shared class permissions before they can share classes.

You add shared class permissions to the `java.policy` file using the ClassLoader class name (wildcards are permitted) and either "read", "write", or "read,write" to determine the access granted. For example:

```
permission com.ibm.oti.shared.SharedClassPermission
        "com.abc.customclassloaders.*", "read,write";
```

If a ClassLoader does not have the correct permissions, it is prevented from sharing classes. You cannot change the permissions of the default bootstrap, application, or extension class loaders.

Adapting custom 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 182. 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 Javadoc document for this package is provided with the SDK in the `docs/content/apidoc` directory.

See "Using the Java Helper API" on page 398 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 239

Chapter 8. Security Guide

An overview of the security features in the IBM SDK for Java.

General information about IBM security providers

Overview of the IBM security providers.

The IBM SDK v7 has been tested with the following default security providers:

- security.provider.1=com.ibm.jsse2.IBMJSSEProvider2
- security.provider.2=com.ibm.crypto.provider.IBMJCE
- security.provider.3=com.ibm.security.jgss.IBMJGSSProvider
- security.provider.4=com.ibm.security.cert.IBMCertPath
- security.provider.5=com.ibm.security.sasl.IBMSASL
- security.provider.6=com.ibm.xml.crypto.IBMXMLCryptoProvider
- security.provider.7=com.ibm.xml.enc.IBMXMLEncProvider
- security.provider.8=com.ibm.security.jgss.mech.spnego.IBMSPNEGO
- security.provider.9=sun.security.provider.Sun

You can also add other IBM security providers, such as the IBMJCEFIPS provider (com.ibm.crypto.fips.provider.IBMJCEFIPS), using one of these methods:

1. To add a new provider statically, edit a Java security properties file. For example, the `java.security` file.
2. To add a new provider from within your application, use the methods of the java.security.Security class. For example:

   ```
   java.security.Security.addProvider()
   ```

Hardware Cryptographic support

The IBM SDK for AIX, v7 supports the IBM PCI-X Cryptographic Coprocessor.

The IBM JSSE2 provider can use these adapters through the IBMPKCS11impl provider.

Note:

1. You can install more than one adapter in any given workstation.
2. If you install more than one adapter, its driver determines how each card is assigned to slots.
3. When a slot number is not specified, 0 is assigned.

The IBM SDK for AIX, v7 might support older adapters that are no longer available. For more information, see the IBMPKCS11 supported devices list, https://www.ibm.com/developerworks/java/jdk/security/70/secguides/pkcs11implDocs/IBMPKCS11SupportList.html.

© Copyright IBM Corp. 2011

IBM PCI-X Cryptographic Coprocessor (PCIXCC)

The IBM PCI-X Cryptographic Coprocessor adapter adds a high-security environment to your AIX server systems for cryptographic functions and sensitive custom applications.

For detailed information about this adapter, see the IBM PCI-X Cryptographic Coprocessor's home page at http://www.ibm.com/security/cryptocards/pcixcc/overview.shtml.

Installing the adapter hardware

For installation instructions, go to the IBM PCI-X Cryptographic Coprocessor's home page, click the **Library** link in the panel on the left, and select the **IBM 4764 Installation Manual** link.

Note: The IBM SDK requires no special installation or configuration to use the adapter because it uses the PKCS#11 interface to communicate with the adapter. However, there is an adapter specific PKCS11 configuration file that must be used in order to use supported adapters. For more information, refer to the `PKCS11ImplConfigSamples.jar` file available from http://www.ibm.com/developerworks/java/jdk/security/70/.

iKeyman tool

The iKeyman utility is a tool for key databases containing digital certificates and keys.

With iKeyman, you can:

- Create and manage key databases.
- Create self-signed digital certificates for testing.
- Add certificate authority (CA) and intermediate certificates.
- Transfer certificates between key databases.
- Create certificate requests and receive a digital certificate issued by a CA in response to a request.
- Create, import and export symmetric keys.

History of changes

Version 7

No changes.

Version 6

Improved error messages and command help.

PKCS#11 access provided by IBMPKCS11Impl Provider rather than IBMPKCS11 Provider and a native library.

Support for symmetric key management (only supported by JCEKS and PKCS#11 key databases).

Use of the Java CMS Provider for CMS key databases rather than a native library.

Version 1.4.1

Added an iKeyman wrapper that starts the correct tool class.

Documentation

For more information, including information about the iKeyman GUI, see the *iKeyman User Guide* at: http://www.ibm.com/developerworks/java/jdk/security/index.html.

Java Authentication and Authorization Service (JAAS) V2.0

The Oracle Corporation Java platform provides a means to enforce access controls based on *where* code came from and *who signed* it. These access controls are needed because of the distributed nature of the Java platform where, for example, a remote applet can be downloaded over a public network and then run locally.

However, before SDK v1.4.0, the Java platform did not provide a way to enforce similar access controls based on *who runs* the code. To provide this type of access control, the Java security architecture requires the following:

- Additional support for authentication (determining who is running the code)
- Extensions to the existing authorization components to enforce new access controls based on who was authenticated

The Java Authentication and Authorization Service (JAAS) framework provides these enhancements.

For a general overview of JAAS, see the Oracle Web site: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html.

Differences between IBM and Oracle versions of JAAS

IBM implementations are contained in the com.ibm.* package instead of the com.sun.* package.

Further reading

For detailed information, including API documentation and samples, see the developerWorks Web site at http://www.ibm.com/developerworks/java/jdk/security/index.html. This site contains the *LoginModule Developer's Guide* and sample code in "HelloWorld.tar".

History of changes

A history of the changes to the Java Authentication and Authorization Service (JAAS) since it was added to the SDK.

Changes for Version 7

No changes.

Changes for Version 6

Added a new JAAS login module that enables users to perform authentication using credentials stored in an LDAP directory service. The new module provides LDAP support for existing JAAS-enabled applications.

Original release

The original release of JAAS for AIX and the Java Platform included the following login module and principal classes:

- com.ibm.security.auth.module.AIXLoginModule
- com.ibm.security.auth.AIXPrincipal
- com.ibm.security.auth.AIXNumericGroupPrincipal
- com.ibm.security.auth.AIXNumericUserPrincipal

These original platform-dependent principal classes will be replaced by a set of platform-independent principal classes in future releases of JAAS for AIX. To ease migration, this version of JAAS contains the original set as well as the new set of principal classes. Also included is a new login module called com.ibm.security.auth.module.AIXLoginModule2000, which has the same function as AIXLoginModule but references the new set of principals. Additional principal classes have been included to facilitate the writing of new login modules.

You are encouraged to use the new set of principals when developing applications that use JAAS. Previously developed applications will be compatible with this version as well as future versions of JAAS released for the SDK v1.4.0.

If migrating applications to the new set of principals is desired, then most changes encountered will be in JAAS policy and configuration files rather than in the applications. Refer to the following table for guidance.

Table 13. New class names

Original class	Replaced by
AIXPrincipal	UsernamePrincipal
AIXNumericGroupPrincipal	GroupIDPrincipal PrimaryGroupIDPrincipal
AIXNumericUserPrincipal	UserIDPrincipal
n/a	DomainPrincipal
n/a	DomainIDPrincipal
n/a	ServerPrincipal
n/a	WkstationPrincipal
AIXLoginModule	AIXLoginModule2000
AIX64LoginModule	(no change)

Principal classes are found in the com.ibm.security.auth package while the login module is found in the com.ibm.security.auth.module package. Check the JAAS API documentation (Javadoc information) for more information on the new principal classes.

For example, this JAAS policy grant block:

```
grant Principal com.ibm.security.auth.AIXPrincipal "bob",
      Principal com.ibm.security.auth.AIXNumericUserPrincipal
           "727",
      Principal com.ibm.security.auth.AIXNumericGroupPrincipal
           "12" {
   permission java.util.PropertyPermission "java.home", "read";
};
```

is replaced by:

```
grant Principal com.ibm.security.auth.UsernamePrincipal "bob",
      Principal com.ibm.security.auth.UserIDPrincipal "727",
      Principal com.ibm.security.auth.GroupIDPrincipal "12" {
   permission java.util.PropertyPermission "java.home", "read";
};
```

Java Certification Path (CertPath)

The Java Certification Path API provides interfaces and abstract classes for creating, building, and validating certification paths (also known as "certificate chains").

Differences between IBM and Oracle versions of CertPath

The IBM CertPath classes differ from the Oracle version in the following ways:

- The IBM CertPath provider is in the package com.ibm.security.cert.
- The IBM CertPath provider is called "IBMCertPath". Oracle does not have a separate provider for CertPath.
- To enable CRL Distribution Points extension checking, use the system property **com.ibm.security.enableCRLDP**. The system property used by the Oracle version is **com.sun.security.enableCRLDP**.
- When checking the CRL Distribution Points extension of the certificate, the Oracle CertPath provider retrieves the CRL only if the CRL location is specified as an HTTP URL value inside the extension. The IBM provider recognizes both HTTP and LDAP URLs.
- The IBM implementation of CertPath supports the processing of both complete CRLs and delta CRLs. Setting the com.ibm.security.enableDELTACRL system property to `true` enables the use of both delta CRLs and complete CRLs if revocation checking is enabled by the caller. If com.ibm.security.enableDELTACRL is set to false, or is not set, only complete CRLs are used.

Documentation

For detailed information, including API documentation and samples, see the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

History of changes

A history of the changes to CertPath since it was added to the SDK.

Changes for Version 7

- You can now disable weak cryptographic algorithms, such as the MD2 message digest algorithm, which is no longer considered secure. The algorithms you want to disable can be specified with a new security property, **jdk.certpath.disabledAlgorithms**. Any algorithms that you specify as disabled are not allowed in certification path processing. This security property can also be used to place restrictions on key sizes.
- You can now specify that revocation checking is performed only for the end-entity certificate of the certificate chain. This behavior is specified with the Java system property

  `-Dcom.ibm.security.onlyCheckRevocationOfEECert=true`

.

Changes for Version 6

- Enhanced CRL validation and CRL processing to more closely comply with the PKIX Certificate and CRL Profile (RFC 3280) Section 6.3, entitled “CRL Validation”. This processing is performed only if revocation checking is enabled by the caller inside the supplied PKIXParameters object.
- Added the com.ibm.security.enableDELTACRL system property to use both delta CRLs and complete CRLs if revocation checking is enabled by the caller. If com.ibm.security.enableDELTACRL is set to false, or is not set, only complete CRLs are used.
- Added the com.ibm.security.enableAIAEXT system property to use LDAP URIs found in any Authority Information Access extensions in certificates on the certificate path. For each LDAP URI found, an LDAPCertStore object is created and added to the collection of CertStores used to locate other certificates needed to build the certificate path.

Changes for Version 5.0

- Added support for checking a certificate's revocation status based on On-Line Certificate Status Protocol (OCSP).
- Added a new constructor and a public API in the TrustAnchor class:

```
public TrustAnchor(X500Principal caPrincipal, PublicKey pubKey, byte[]
                        nameConstraints);
public final X500Principal getCA();
```

- Added new public APIs in X509CertSelector:

```
public X500Principal getIssuer();
public void setIssuer(X500Principal issuer);
public X500Principal getSubject();
public void setSubject(X500Principal subject);
```

- Added new public APIs in X509CRLSelector:

```
public void setIssuers(Collection issuers);
public void addIssuer(X500Principal issuer);
public Collection getIssuers();
```

- Changed the PolicyQualifier class to non-final. The public APIs have changed to be final.

Changes for Version 1.4.2

- Improved the performance of the IBM CertPath provider.
- Added limited support for the CRL Distribution Points extension.
- Added caching to cache lookups in the IBM LDAP CertStore.

Changes for Version 1.4.1, Service Refresh 1

- The trusted certificate that acts as TrustAnchor can be an X.509 v1 certificate.
- When you specify the certificate's subject or issuer name as a String in X509CertSelector, the search for a matched certificate mechanism checks only the name value and ignores the tag type.

Changes for Version 1.4.0

- Certificates from CertificatePair entry can be retrieved from LDAP type certstore.
- Changed the framework package name from javax.security.cert to java.security.cert. The old framework package is still supported.

Java Cryptography Extension (JCE)

The Java Cryptography Extension (JCE) provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects. JCE supplements the Java platform, which already includes interfaces and implementations of message digests and digital signatures.

You can obtain unrestricted jurisdiction policy files from http://www.ibm.com/developerworks/java/jdk/security/index.html. The policy files are replaced with restricted policy files when you upgrade your SDK. Before upgrading your SDK, make a backup of your policy files. After upgrading your SDK, install your backup policy files if you need an unrestricted policy.

The v1.4.2 unrestricted (and restricted) jurisdiction policy files are suitable for use with v5.0 and later. The v1.4.1 files are not suitable.

Differences between IBM and Oracle versions of JCE

The com.sun.* packages are reimplemented by IBM and renamed com.ibm.* packages.

The IBM version of JCE differs from the Oracle version in the following ways:

- The com.sun.crypto.* packages are reimplemented by IBM and renamed com.ibm.crypto.* packages.
- The IBM JCE provider replaces the Oracle providers sun.security.provider.Sun, com.sun.rsajca.Provider, and com.sun.crypto.provider.SunJCE.
- IBM provides more algorithms than Oracle does:

  Cipher algorithms

  AES

  Blowfish

  DES

  El Gamal

  Mars

  ARCFOUR

  PBE with MD2 and DES

  PBE with MD2 and Triple DES

  PBE with MD2 and RC2

  PBE with MD5 and DES

  PBE with MD5 and Triple DES

  PBE with MD5 and RC2

  PBE with SHA1 and DES

  PBE with SHA1 and TripleDES

  PBE with SHA1 and RC2

  PBE with SHA1 and 40-bit RC2

PBE with SHA1 and 128-bit RC2

PBE with SHA1 and 40-bit RC4

PBE with SHA1 and 128-bit RC4

PBE with SHA1 and 2-key Triple DES

PBE with SHA1 and 3-key Triple DES

RC2

RC4

RSA encryption/decryption

RSA encryption/decryption with OAEP Padding

Seal

Triple DES

Signature algorithms

SHA1 with RSA

SHA2 with RSA

SHA3 with RSA

SHA5 with RSA

MD5 with RSA

MD2 with RSA signature

SHA1 with DSA signature

Message digest algorithms

SHA1

SHA2

SHA3

SHA5

MD5

MD2

Message authentication code (MAC)

Hmac/SHA1

Hmac/MD5

Hmac/SHA2

Hmac/SHA3

Hmac/SHA5

Key agreement algorithm

DiffieHellman

Random number generation algorithms

IBMSecureRandom

IBM SHA1PRNG

Key Store

JCEKS

JKS

PKCS12KS

Documentation

For detailed information, including API documentation and samples, see the developerWorks Web site at http://www.ibm.com/developerworks/java/jdk/security/index.html.

History of changes

A history of the changes to the Java Cryptography Extension (JCE) since it was added to the SDK.

Changes for Version 7

JCE has added support for ECDH and ECDSA algorithms for polynomial curves. Binary curves are not supported.

Changes for Version 6

- Added CipherTextStealing mode.
- Added ISO10126Padding support.
- Added PBKDF2HmacSHA1Factory and PBKDF2KeyImpl.
- Added supporting classes for Tls use with JSSE.

Changes for Version 5.0, Service Refresh 4

- Added the EL Gamal Cipher and supporting classes.

Changes for Version 5.0

- Added RSA with OAEP Padding.
- Added the SHA2withRSA, SHA3withRSA and SHA5withRSA signature algorithms.
- Added the HmacSHA2, HmacSHA3, HmacSH5 MAC algorithms.
- Added the ARCFOUR encryption algorithm.

Changes for Version 1.4.2

- Added the SHA2, SHA3 and SHA5 hashing algorithms.
- Added the SHA1PRNG algorithm for generating pseudo random numbers.

Changes for Version 1.4.0

- Added the AES cipher algorithm.
- Changed to strong cryptography by default. Unlimited cryptography is available.
- Provider authentication of the JCE framework no longer required.
- JCE is now shipped with the Java SDK v1.4 on all platforms.

Known limitations

Known limitations of the IBM JCE provider.

SecureRandom performance

The IBM SecureRandom generator in the IBM JCE provider meets several government security specifications. However, in order to meet these specifications, the authentication process can take longer than the authentication process for other providers. This problem has been seen on WebSphere Application Server 7.0 when connecting to an Oracle database server.

If authentication does not need to be bound by the strict JCE security specification, use the IBM SecureRandom provider instead. This provider is available from Java 6 SR9 onwards. For more information, see "IBM SecureRandom Provider" on page 209.

Java Generic Security Service (JGSS)

Java Generic Security Service (JGSS) API provides secure exchange of messages between communicating applications.

JGSS is an API framework that uses Kerberos V5 as the underlying default security mechanism. The API is a standardized abstract interface under which you can plug different security mechanisms that are based on private-key, public-key, and other security technologies.

JGSS shields secure applications from the complexities and peculiarities of the different underlying security mechanisms. JGSS provides identity and message origin authentication, message integrity, and message confidentiality. JGSS also features an optional Java Authentication and Authorization Service (JAAS) Kerberos login interface, and authorization checks. JAAS augments the access control features of Java, which is based on CodeSource with access controls based on authenticated principal identities.

Differences between the IBM and Oracle versions of JGSS

The IBM version of JGSS differs from the Oracle version in the following ways:

- The com.sun.* packages are reimplemented by IBM and renamed com.ibm.* packages.
- The format of the parameters passed to the Java tools **kinit**, **ktab**, and **klist** is different from the equivalent tools provided in the Oracle version of JGSS.

Documentation

For detailed information about JGSS, including API documentation and samples, see the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

Installing Kerberos

Configuration details for JGSS can be found in the AIX Kerberos configuration file, `krb5.conf`. For information about the contents of this file, and how to configure Kerberos, see the administration guide in the `/usr/lpp/krb5/doc` directory.

For AIX installation instructions for Kerberos, see http://publibn.boulder.ibm.com/doc_link/en_US/a_doc_lib/aixins/aixinsgd/install_kerberos.htm.

The AIX Expansion Pack CD (see http://www.ibm.com/servers/aix/expansionpack/) contains both the client and server versions of the IBM Network

Authentication Service (which implements Kerberos). The Expansion Pack also contains both PDF and HTML versions of the Network Authentication Service Administrator and User Guide. English, French, Spanish, Japanese, Chinese, and Korean versions of this guide are available. The files are named `krb5.doc.<locale>.pdf` or `krb5.doc.<locale>.html`, where <locale> is the language. After installation, the guide is located in a subdirectory of `/usr/lpp/krb5/doc`.

History of changes

A history of the changes to the Java Generic Security Service (JGSS) since it was added to the SDK.

Changes for Version 7

New options are available in the `ktab` Java tool

Support for OK-AS-DELEGATE flag
Credential delegation is enabled when you set the OK-AS-DELEGATE service ticket flag.

Cross-realms for HTTP or SPNEGO
The service principal name is generated from the fully qualified hostname of the web server.

Tickets that do not have addresses
The `no_addresses` and `noaddresses` tags have been moved from the JAAS configuration file to the Kerberos configuration file. The default value is true. These tags are no longer supported in the JAAS configuration file.

`refreshKrb5Config` flag added to JAAS configuration
This flag is defined in the JAAS configuration file. The flag refreshes the Kerberos configuration file.

Accessing an unavailable KDC
The `krb5.kdc.bad.policy` security system property provides smarter access to the KDC, by reducing the number of attempts made to access an unavailable KDC.

Keytab content is always refreshed
The keytab content is always reloaded immediately after the AP-REQ message is received on the acceptor side.

Changes for Version 6

Added CipherTextStealing mode.

Changes for Version 5.0, Service Refresh 1

Added AES as a supported algorithm type
These additional algorthims can be set in the `krb5.conf` file under [libdefault] as follows:

```
default_tkt_enctypes = aes128-cts-hmac-sha1-96
default_tkt_enctypes = aes256-cts-hmac-sha1-96
default_tgs_enctypes = aes128-cts-hmac-sha1-96
default_tgs_enctypes = aes256-cts-hmac-sha1-96

default_checksum = hmac-sha1-96-aes128
default_checksum = hmac-sha1-96-aes256
```

Changes for Version 5.0

TCP or UDP Preference Configuration

Added JSE support for the **udp_preference_limit** property in the Kerberos configuration file `krb5.ini`. When sending a message to the KDC, the JSE Kerberos library uses TCP if the size of the message is above **udp_preference_list**. If the message is smaller than **udp_preference_list**, UDP is tried up to three times. If the KDC indicates that the request is too big, the JSE Kerberos library uses TCP.

IPv6 support in Kerberos

Added JSE support for IPv6 addresses in Kerberos tickets. Before v5.0, only IPv4 addresses were supported in tickets.

TGT Renewals

Added support for Ticket Granting Ticket (TGT) renewal to the Java Authentication and Authorization Service (JAAS) Kerberos login module, Krb5LoginModule. This support allows long-running services to renew their TGTs automatically without user interaction or requiring the services to restart. With this feature, if Krb5LoginModule obtains an expired ticket from the ticket cache, the TGT is automatically renewed and is added to the Subject of the caller who requested the ticket. If the ticket cannot be renewed for any reason, Krb5LoginModule uses the configured callback handler to retrieve a username and password to acquire a new TGT.

To use this feature, configure Krb5LoginModule to use the ticket cache and set the newly introduced **renewTGT** option to true. Here is an example of a JAAS login configuration file that requests TGT renewal:

```
server {
 com.ibm.security.auth.module.Krb5LoginModule required
 principal=principal@your_realm
  useDefaultCcache=TRUE
   renewTGT=true;
};
```

Note that if **renewTGT** is set to `true`, **useDefaultCcache** must also be set to true; otherwise, it results in a configuration error.

Changes for Version 1.4.2

Configurable Kerberos Settings

You can provide the name and realm settings for the Kerberos Key Distribution Center (KDC) either from the Kerberos configuration file, or by using the system properties files `java.security.krb5.kdc` and `java.security.krb5.realm`. You can also specify the boolean option **refreshKrb5Config** in the entry for Krb5LoginModule in the JAAS configuration file. If you set this option to `true`, the configuration values are refreshed before the login method of the Krb5LoginModule is called.

Added support for Slave Kerberos Key Distribution Center

Kerberos uses slave KDCs so that, if the master KDC is unavailable, the slave KDCs respond to your requests. In previous releases, Kerberos tried the master KDC only and gave up if there was no response in the default KDC timeout period.

Added support for TCP for Kerberos Key Distribution Center Transport

Kerberos uses UDP transport for ticket requests. In cases where Kerberos tickets exceed the UDP packet size limit, Kerberos supports automatic

fallback to TCP. If a Kerberos ticket request using UDP fails and the KDC returns the error code `KRB_ERR_RESPONSE_TOO_BIG`, TCP becomes the transport protocol.

Kerberos Service Ticket in the Private Credentials of the Subject
The Kerberos service ticket is stored in the private credentials of the Subject. This gives you access to the service ticket so that you can use the ticket outside the JGSS, for example in native applications or for proprietary uses. In addition, you can reuse the service ticket if the application tries to establish a security context to the same service again. The service ticket must be valid for it to be reusable.

Changes for Version 1.4.1

- Added wrappers for the **`klist`**, **`kinit`**, and **`ktab`** Java tools. These wrappers call the relevant tool classes so that you do not have to remember the full package name.

IBMJSSE2 Provider

The Java Secure Socket Extension (JSSE) is a Java package that enables secure internet communications. The package implements a Java version of SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols. It includes functions for data encryption, server authentication, message integrity, and optional client authentication.

By abstracting the complex underlying security algorithms and "handshaking" mechanisms, JSSE minimizes the risk of creating subtle but dangerous security vulnerabilities. Also, it simplifies application development by serving as a building block that you can integrate directly into your applications. Using JSSE, you can provide for the secure passage of data between a client and a server running any application protocol (such as HTTP, Telnet, NNTP, and FTP) over TCP/IP.

The FIPS provider included with the SDK is undergoing certification with the US Government. The certification progress is available on the CSRC Web site: http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140InProcess.pdf.

Documentation

For detailed information, including API documentation and samples, see http://www.ibm.com/developerworks/java/jdk/security/index.html.

Differences between the IBMJSSE Provider and the IBMJSSE2 Provider

The IBMJSSE2 Provider, which was introduced in the v1.4.2 SDK, has replaced the IBMJSSE Provider. Although they are nearly equivalent, there are differences between the two providers.

The now-discontinued IBMJSSE Provider and the IBMJSSE2 Provider differ in the following ways:

- The IBMJSSE2 Provider is called com.ibm.jsse2.IBMJSSEProvider2.
- The HTTPS protocol handler for the IBMJSSE2 Provider is called com.ibm.net.ssl.www2.protocol.Handler. The com.ibm.net.ssl.internal.www.protocol.Handler and the com.ibm.net.ssl.www.protocol.Handler protocol handlers have been removed.

- The IBMJSSE2 Provider does not support the com.ibm.net.ssl framework. Use the javax.net.ssl framework instead.
- The IBMJSSE2 Provider does not support the SSL version 2 protocol. However, the server side of a JSSE2 connection does accept the SSLv2Hello protocol.
- The AES_256 ciphers require the installation of the JCE Unlimited Strength Jurisdiction Policy. The old IBMJSSE Provider did not use JCE for its cryptographic support and therefore did not require these files.
- The IBMJSSE2 Provider requires a JCE Provider for its cryptography.
- The IBMJSSE2 Provider does not build the server's private key certificate chain from the trusted keystore. The trusted certificates must be added to the server's private key to complete the chain. This is an incompatible change.
- The IBMJSSE2 Provider considers a certificate trusted if you have the private key.
- The HTTPS protocol handler for the IBMJSSE2 Provider performs hostname verification and rejects requests where the host to connect to and the server name from the certificate do not match. A HostnameVerification implementation called com.ibm.jsse2.HostnameVerifierIgnore is provided. com.ibm.jsse2.HostnameVerifierIgnore always accepts the connection even when a mismatch occurs.
- Tracing no longer requires a separate debug jar.
- The class com.ibm.jsse.SSLContext, which in IBMJSSE is used to access secure tokens, has been removed. Use the hardware crypto support in IBMJSSE2 instead. See the documentation on the developerWorks Web site http://www.ibm.com/developerworks/java/jdk/security/index.html for details.
- The IBMJSSEFIPS Provider has been removed. JSSE FIPS support is supported in the IBMJSSE2 Provider and no separate jar is required. See the documentation on the developerWorks Web site http://www.ibm.com/developerworks/java/jdk/security/index.html for instructions how to set up JSSE to run in FIPS mode.

Differences between the IBMJSSE2 Provider and the Oracle version of JSSE

Although they are nearly equivalent, there are differences between the IBMJSSE2 Provider and the Oracle JSSE Provider.

The IBMJSSE2 Provider differs from the Oracle JSSE in the following ways:

- The IBM JSSE Provider is called com.ibm.jsse2.IBMJSSEProvider2.
- The IBM KeyManagerFactory is called IbmX509 or NewIbmX509.
- The IBM TrustManagerFactory is called IbmX509 or IbmPKIX.
- The IBM HTTPS protocol handler is called com.ibm.net.ssl.www2.protocol.Handler.
- IBMJSSE2 does not support the com.sun.net.ssl framework; use the javax.net.ssl framework instead.
- You can use PKIK revocation checking by setting the system property *com.ibm.jsse2.checkRevocation* to “true”.
- The IBM implementation supports the following protocols for the engine class SSLContext, for the API setEnabledProtocols in the SSLSocket, and for SSLServerSocket classes:
  - SSL
  - SSLv3

– TLS
– TLSv1
– SSL_TLS

The IBM implementation *does not* support the “SSLv2Hello”protocol. The IBM implementation supports the SSL v2 protocol. You can use the IBM SSLContext `getInstance()` factory method to control which protocols are enabled for an SSL connection. Using the SSLContext `getInstance()` or the `setEnabledProtocols()` methods provides the same result. With the Oracle version of JSSE, the protocol is controlled through `setEnabledProtocols()`.

- IBM and Oracle support different cipher suites.
- The IBM JSSE TrustManager does not allow anonymous ciphers. To handshake with an anonymous cipher, a custom TrustManager that allows anonymous ciphers must be provided.
- When a null KeyManager is passed to SSLContext, the IBM JSSE KeyManagerFactory implementation first checks the system properties. Next, if it exists, jssecacerts is checked. Finally, the `cacerts` file is used to find the key material. The Oracle version of JSSE creates an empty KeyManager.
- The IBM JSSE X509TrustManager and X509KeyManager throws an exception if any one of three conditions are true:
  – The TrustStore or KeyStore specified by the system properties does not exist.
  – The password is incorrect.
  – The keystore type is inappropriate for the actual keystore.

  The Oracle version of X509TrustManager creates a default TrustManager or KeyManager with an empty keystore.
- The IBM JSSE implementation verifies the entire server or client certificate chain, including trusted certificates. For example, if a trusted certificate has expired, the handshake fails, even though the expired certificate is trusted. The Oracle version of JSSE verifies the certificate chain up to the trusted certificate. Verification stops when it reaches a trusted certificate and the trusted certificate and beyond are not verified.
- The IBM JSSE implementation returns the same set of supported ciphers for the methods `getDefaultCiphersSuites()` and `getSupportedCipherSuites()`. The Oracle version of `getDefaultCipherSuites()` returns the list of cipher suites that provide confidentiality protection and server authentication (that is, no anonymous cipher suites). Oracle's `getEnabledCipherSuites()` returns the entire list of cipher suites that Oracle supports.
- For the Oracle implementation, DSA server certificates can use only *_DH*_* cipher suites. For the IBM implementation, if the server has a DSA certificate only, and only RSA* ciphers are enabled, the connection succeeds with an RSA cipher. DSA will be used for authentication and ephemeral RSA will be used for the key exchange.
- To use a hardware keystore or truststore on an IBM hardware crypto provider with system properties, set the **`javax.net.ssl.keyStoreType`** and **`javax.net.ssl.trustStoreType`** system properties to “PKCS11IMPLKS”. For the Oracle implementation, the system property is set to “PKCS11”.
- The IBM and Oracle JSSE Providers can be enabled to run in FIPS mode although they are enabled differently.

History of changes

A history of the changes to the IBMJSSE2 Provider since it was added to the SDK.

Changes for Version 7

TLS 1.1
The IBMJSSE2 provider now supports Transport Layer Security (TLS) 1.1 as described in RFC 4346. The most important update is protection against cipher block chaining (CBC) attacks.

TLS 1.2
The IBMJSSE2 provider now supports TLS 1.2 as described in RFC 5246. This RFC specifies different internal hashing algorithms, adds new cipher suites, and contains improved flexibility, particularly for negotiation of cryptographic algorithms.

Weak cipher suites deprecated
Some cipher suites have been made obsolete and should not be used. These changes are described in RFC 4346, RFC 5246, and RFC 5469. The obsolete suites are disabled by default in IBMJSSE2. For further details, consult the cipher suite lists in the IBMJSSE2 Provider Reference Guide, https://www.ibm.com/developerworks/java/jdk/security/70/secguides/jsse2Docs/JSSE2RefGuide.html.

Connection-sensitive trust management
Both trust managers and key managers can now examine parameters of the TLS connection, specifically the SSLSession under construction, during the handshake. For example, a trust manager might restrict the types of certificates used, based on the list of valid signature algorithms.

Endpoint verification
An endpoint identification algorithm can be specified to verify that the host address of a remote computer matches the supplied certificate. Although this type of verification is already available for the HTTPS protocol (see HttpsURLConnection and HostnameVerifier), you can now use this verification process at the TLS level.

Algorithm disabling
Weak cryptographic algorithms can now be disabled by specifying system properties.

Server Name Indication (SNI) for JSSE client
The Java SE 7 release supports the Server Name Indication (SNI) extension in the JSSE client. RFC 4366 describes how SNI enables TLS clients to connect to virtual servers.

Elliptic Curve Cryptographic (ECC) Cipher Suites
For details, consult the cipher suite lists in the IBMJSSE2 Provider Reference Guide, https://www.ibm.com/developerworks/java/jdk/security/70/secguides/jsse2Docs/JSSE2RefGuide.html.

Restriction on running IBM JSSE2 in FIPS mode
IBM JSSE2 cannot be run in FIPS mode, because the IBMJCEFIPS provider that is used by IBM JSSE2 is not currently certified for Version 7.

Change for Version 6, Service Refresh 9

- IBM JSSE2 allows SSL V3 or TLS V1 session renegotiation with peers that have implemented RFC 5746: Transport Layer Security (TLS) – Renegotiation Indication Extension. For more information, see: "IBMJSSE2 support for RFC 5746 Transport Layer Security (TLS) – Renegotiation Indication Extension" on page 201

Changes for Version 6

- Added default SSLContext methods to SSLContext.
- Added the new SSLParameters class.
- Removed plugability restrictions from the JSSE Framework.

Changes for Version 5.0, Service Refresh 5

- When IBMJSSE2 is used as a server, if the SSLv3 protocol is to be used for the handshake, it will no longer agree to use any of the AES cipher suites. Previously, the selection of the cipher suite was independent of the protocol selected so you could do an old-style SSLv3 handshake with a more modern AES cipher suite. The TLS protocol is not affected by this change. This change was required to support Microsoft Vista clients.

Changes for Version 5.0

- Removed the IBMJSSE Provider. Use the IBMJSSE2 Provider instead.
- The IBMJSSE2 implementation supports only IBM hardware cryptographic providers. Applications that ran in v1.4.2 using the IBMPKCS11Impl provider will now be required to use a configuration file in order to run successfully.
- SSLEngine (non-blocking I/O) allows SSL/TLS applications to choose their own I/O and compute models.
- Enhanced TrustManager support for HTTP/HTTPS.
- New and updated Methods and Classes.
- Kerberos cipher suites are available, if supported by the operating system.

Changes for Version 1.4.2.

The IBMJSSE2 Provider is new for Version 1.4.2.

IBMJSSE2 support for RFC 5746 Transport Layer Security (TLS) – Renegotiation Indication Extension

IBM JSSE2 allows SSL V3 or TLS V1 session renegotiation with peers that have implemented RFC 5746.

The IETF has published RFC 5746 Transport Layer Security (TLS) – Renegotiation Indication Extension. RFC 5746 defines a mechanism to implement TLS/SSL handshake renegotiation securely. Use of RFC 5746 replaces the industry wide interim solution of disabling all renegotiation implemented after the weakness was discovered.

After applying this APAR, IBM JSSE2 allows SSL V3 or TLS V1 session renegotiation with peers that have implemented RFC 5746. Session renegotiation with peers that do not support RFC 5746 reverts to the interim disablement solution. By default, unsecured renegotiation is not allowed. Use the system property **com.ibm.jsse2.renegotiate** to control how unsecured negotiations are handled by IBM JSSE2.

Read RFC 5746 for additional details if interested in the underlying TLS protocol changes to correct the weakness.

The following system properties are available to control how restrictive IBM JSSE2 is in the enforcement of RFC 5746:

com.ibm.jsse2.extended.renegotiation.indicator=[BOTH | CLIENT | OPTIONAL | SERVER]

Use this property to force all negotiations to require RFC 5746, not just renegotiations. This negotiation would only be practical after all the required communication partners have implemented RFC 5746. The default setting is **OPTIONAL**.

com.ibm.jsse2.extended.renegotiation.indicator=BOTH

Causes the IBM JSSE2 Server or IBM JSSE2 client to connect only if the peer indicated support for RFC 5746 renegotiation.

Note: Setting the property to **BOTH** causes interoperability problems with clients or servers that have not been updated to support RFC 5746.

com.ibm.jsse2.extended.renegotiation.indicator=CLIENT

Causes the IBM JSSE2 Client to connect only if the server indicated support for RFC 5746 Renegotiation.

Note: Setting the property to **CLIENT** causes interoperability problems with servers that have not been updated to support RFC 5746.

com.ibm.jsse2.extended.renegotiation.indicator=OPTIONAL

This setting is the default. Using this option means that the IBM JSSE2 Server or IBM JSSE2 Client do not require the renegotiation indicator during the initial handshake.

com.ibm.jsse2.extended.renegotiation.indicator=SERVER

Causes the IBM JSSE2 Server to connect only if the client indicated support for RFC 5746 Renegotiation.

Note: Setting the property to **SERVER** causes interoperability problems with clients that have not been updated to support RFC 5746.

com.ibm.jsse2.renegotiate = [ABBREVIATED | ALL | DISABLED | NONE]

Use this property to change the renegotiation ability of IBM JSSE2. The default value is **NONE**.

com.ibm.jsse2.renegotiate=ABBREVIATED

This setting overrides and allows unsecured abbreviated handshake during renegotiation when session continuity is proven. RFC 5746 renegotiations are allowed.

com.ibm.jsse2.renegotiate=ALL

This setting overrides and allows unsecured full handshake, and unsecured abbreviated handshake, during renegotiation. RFC 5746 renegotiations are allowed.

com.ibm.jsse2.renegotiate=DISABLED

This setting overrides and disables all unsecure and RFC 5746 renegotiations.

com.ibm.jsse2.renegotiate=NONE

This setting is the default. No unsecured handshake renegotiation is allowed. Only RFC 5746 renegotiations are allowed.

com.ibm.jsse2.renegotiation.peer.cert.check=[OFF | ON]

Use this property to change the renegotiation ability of IBM JSSE2 to

require the peer support specified in RFC 5746. This requirement is only practical after all the required communication partners have implemented RFC 5746. The default value is **OFF**.

com.ibm.jsse2.renegotiation.peer.cert.check=OFF
This setting is the default. It stops the IBM JSSE2 Client or IBM JSSE2 Server performing an identify check against the certificate from the peer. The result is to allow the peer certificate to change during renegotiation.

com.ibm.jsse2.renegotiation.peer.cert.check=ON
This setting causes the IBM JSSE2 Client or IBM JSSE2 Server to perform a comparison against the certificate from the peer. The reason is to ensure that the certificate does not change during renegotiation. The comparison is applicable to both secure and non-secure renegotiations.

IBMPKCS11Impl Provider

The IBMPKCS11Impl Provider uses the Java Cryptography Extension (JCE) and Java Cryptography Architecture (JCA) frameworks to add the ability to use hardware cryptography through the Public Key Cryptographic Standards #11 (PKCS #11) standard.

This provider takes advantage of hardware cryptography in the existing JCE architecture. It gives Java programmers the significant security and performance advantages of hardware cryptography with minimal changes to existing Java applications. Because the complexities of hardware cryptography are handled in the normal JCE, advanced security and performance using hardware cryptographic devices is available readily.

PKCS#11 is a standard that provides a common application interface to cryptographic services on various platforms through several hardware cryptographic devices. See the IBMPKCS11Impl provider user guide for a list of supported devices.

Remember: Use the correct PKCS11 configurations files for Version 7 of the SDK. You can download sample configuration files from http://www.ibm.com/developerworks/java/jdk/security/70/.

Differences between IBM and Oracle versions of IBMPKCS11Impl

- The Oracle keystore is named PKCS11 and the IBM keystore is called IBMPKCS11KS.
- The Oracle keystore requires that all trusted certificates have the attribute **CKA_TRUSTED** set to true. The IBM keystore assumes that any certificates on the device are trusted. The IBM keystore can work with data that was saved using the Oracle keystore. However, the Oracle keystore might not be able to work with data saved using the IBM keystore.

Documentation

For detailed information, including API documentation, see the developerWorks Web site at http://www.ibm.com/developerworks/java/jdk/security/index.html.

History of changes

A history of the changes to the IBMPKCS11Impl Provider since it was added to the SDK.

Changes for Version 7

No changes.

Changes for Version 6

- Added support for TLS/SSL key generations and ECDSA signature algorithms including SHA1withECDSA, SHA256withECDSA, SHA384withECDSA, and SHA512withECDSA.
- Added session management so that applications can concurrently execute multi-part crypto operations within a single provider.

Changes for Version 5.0

Updated IBMPKCS11Impl to allow more algorithms and to allow the Oracle 5.0 methods of initialization of the provider. The new algorithms are:

- AES
- Diffie-Hellman
- RC4, also known as ArcFour
- Blowfish
- SHA-256
- SHA-384
- SHA-512
- SHA256withRSA
- SHA384withRSA
- SHA512withRSA
- HmacMD5
- HmacSHA1
- HmacSHA256
- HmacSHA384
- HmacSHA512

Added the ability to pass in a configuration file to the provider. This configuration file can contain a significant amount of information about the device; for example, what it should or should not do. After the provider is created, the application can log in to the card in different ways. Some devices allow you to perform some cryptographic functions without logging into the device. The v1.4.2 ways to initialize the device still work. However, you can no longer have more than one of these providers at a time. Instead, with this release, you can initialize more than one IBMPKCS11Impl provider using the 5.0 configuration file and login methods.

Deprecated the classes DESPKCS11KeyParameterSpec and DESedePKCS11KeyParameterSpec. Use the GeneralPKCS11KeyParameterSpec class for all symmetric key types (for instance, DES, DESede, AES, RC4, Blowfish).

Changes for Version 1.4.2

The IBMPKCS11Impl Provider was new for v1.4.2.

IBMJCEFIPS Provider

The IBM Java JCE (Java Cryptographic Extension) FIPS Provider (IBMJCEFIPS) for multi-platforms is a scalable, multi-purpose cryptographic module that supports FIPS-approved cryptographic operations through Java APIs.

IBMJCEFIPS was introduced in Version 1.4.2.

The IBMJCEFIPS includes the following Federal Information Processing Standards (FIPS) 140-2 [Level 1] compliant components:

- IBMJCEFIPS for AIX

To meet the requirements specified in the FIPS publication 140-2, the encryption algorithms used by the IBMJCEFIPS Provider are isolated into the IBMJCEFIPS Provider cryptographic module. You can access the module using the product code from the Java JCE framework APIs. Because the IBMJCEFIPS Provider uses the cryptographic module in an approved manner, the product complies with the FIPS 140-2 requirements.

Type	Algorithm	Specification
Symmetric Cipher	AES (ECB, CBC, OFB, CFB, and PCBC)	FIPS 197
Symmetric Cipher	Triple DES (ECB, CBC, OFB, CFB, and PCBC)	FIPS 46-3
Message Digest	SHA1 SHA-256 SHA-384 SHA-512 HMAC-SHA1	FIPS 180-2 FIPS 198a
Random Number Generator	FIPS 186-2 appendix 3.1	FIPS 186-2
Digital Signature	DSA (512 - 1024)	FIPS 186-2
Digital Signature	RSA (512 – 2048)	FIPS 186-2

In addition, the IBMJCEFIPS supports the following unapproved algorithms:

Type	Algorithm	Specification
Asymmetric Cipher	RSA	PKCS#1
Key Agreement	Diffie-Hellman	PKCS #3 (Allowed in Approved mode)
Digital Signature	DSAforSSL	Allowed for use inside the TLS protocol
Digital Signature	RSAforSSL	Allowed for use inside the TLS protocol
Message Digest	MD5	FIPS 180-2
Random Number Generation	Universal Software Based Random Number Generator	Available upon request from IBM. Patented by IBM, EC Pat. No. EP1081591A2, U.S. pat. Pend.

Important: The com.ibm.crypto.fips.provider.IBMJCEFIPS class does not include a keystore (such as JKS or JCEKS) because of FIPS requirements and algorithms.

Therefore, if you are using com.ibm.crypto.fips.provider.IBMJCEFIPS and require JKS, you must specify the com.ibm.crypto.provider.IBMJCE in the provider list.

For more detailed information about the FIPS certified provider IBMJCEFIPS, see the *IBM Java JCE FIPS 140-2 Cryptographic Module Security Policy*. For usage information and details of the API, see the *IBM Java JCE FIPS (IBMJCEFIPS) Cryptographic Module API* document.

Differences between IBM and Oracle versions of IBMJCEFIPS

Oracle does not provide IBMJCEFIPS.

Documentation

For detailed information, including API documentation and Security Policy, see the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

IBM SASL Provider

Simple Authentication and Security Layer, or SASL, is an Internet standard (RFC 2222) that specifies a protocol for authentication and optional establishment of a security layer between client and server applications. SASL defines how authentication data is to be exchanged but does not itself specify the contents of that data. It is a framework into which specific authentication mechanisms that specify the contents and semantics of the authentication data can fit.

The Java SASL API defines classes and interfaces for applications that use SASL mechanisms. It is defined to be mechanism-neutral: the application that uses the API does not need to be hardwired into using any particular SASL mechanism. The API supports both client and server applications. It allows applications to select the mechanism to use based on preferred security features, such as whether they are susceptible to passive dictionary attacks or whether they accept anonymous authentication. The Java SASL API also allows developers to use their own, custom SASL mechanisms. SASL mechanisms are installed by using the Java Cryptography Architecture (JCA).

The IBM SASL provider supports the following client and server mechanisms.

Client mechanisms

- PLAIN (RFC 2595). This mechanism supports cleartext username/password authentication.
- CRAM-MD5 (RFC 2195). This mechanism supports a hashed username/password authentication scheme.
- DIGEST-MD5 (RFC 2831). This mechanism defines how HTTP Digest Authentication can be used as a SASL mechanism.
- GSSAPI (RFC 2222). This mechanism uses the GSSAPI for obtaining authentication information. It supports Kerberos v5 authentication.
- EXTERNAL (RFC 2222). This mechanism obtains authentication information from an external channel (such as TLS or IPsec).

Server mechanisms

- CRAM-MD5
- DIGEST-MD5
- GSSAPI (Kerberos v5)

Differences between the Oracle and IBM SASL Providers

Only the package names, for example com.ibm.security.sasl, and the provider name are different from the Oracle Implementation: com.ibm.security.sasl.IBMSASL.

History of changes

Version 7
NTLM support is added as a Java SASL mechanism for use with other protocols.

Version 6
No changes.

Version 5.0
The IBM SASL Provider is new for v5.0.

Documentation

Detailed information, including API documentation and samples, is on the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

IBM Key Certificate Management

Uses of the IBM Key Certificate Management utilities.

The Key Certificate Management utilities make up a set of packages used to:

- Access keys and certificates stored in any format.
- Extract information from a KeyStore, given a Subject Key Identifier (SKI) and a set of certificate generation APIs, to create a self-signed certificate.
- Generate a CertificateRequest.
- Obtain a certificate signed by a CA.

The Key Certificate Management utilities can:

- Generate a CertificateRequest, and submit the request to a CA using the Java Public Key Infrastructure (PKI) to sign a certificate and then receive the signed certificate.
- Generate a PKCS10 request.
- Generate a Self-Signed Certificate.
- Revoke a signed certificate from a CA using the Java PKI.
- Import certificates from the input stream to the KeyStore or export certificates from the KeyStore to the output stream.
- Copy a keystore from one keystore format to another keystore format.
- Extract information from a KeyStore given a Subject Key Identifier.

The Subject Key Identifier is specified in RFC 3820, Section 4.2.1.2, http://www.faqs.org/rfcs/rfc3820.html.

History of changes

Version 7
No changes.

Version 6
No changes.

Version 5.0, Service Refresh 1
The Key Certificate Management utility is new for Version 5.0, Service Refresh 1.

Documentation

The *Key Certificate Management How-to Guide* and Javadoc information are on the developerWorks Web site, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

Keytool

Keytool is a key and certificate management utility for managing public and private key pairs and associated certificates.

You can use Keytool to administer your public and private key pairs and associated certificates. These keys and certificates can be used for self-authentication, or data integrity and authentication services.

Keytool can also administer secret keys that are used in symmetric encryption/decryption, for example, DES.

Keytool stores the keys and certificates in a keystore.

History of changes

Version 7
The Keytool utility has been updated with increased support for certificates and Certificate Revocation Lists (CRLs).

The Keytool utility has support for generating EllipticCurve keys and certificates.

Version 6
The command interface for Keytool is changed for this release.

Documentation

The *Keytool - Key and Certificate Management Tool Guide* and Javadoc information are on developerWorks, at http://www.ibm.com/developerworks/java/jdk/security/index.html.

Java XML Encryption and signatures

Java XML Encryption provides a standard set of APIs for XML digital encryption services and digital signature services.

You can use XML Encryption to perform fine-grained, element-based encryption of fragments in an XML Document. You can encrypt arbitrary binary data and include it in an XML document. The result of encrypting data is an XML Encryption element that contains or references the cipher data.

The IBM provider implementation of Java XML Encryption supports these algorithms:

- Block encryption: triple DES, AES-128 and AES-256
- Key transport: RSA-v1.5 and RSA-OAEP
- Symmetric key wrap: triple DES, AES-128 and AES-256

- Transformation: Base64, XPath, and XSLT

You can use XML Digital Signature to perform detached, enveloped, and enveloping signatures and to sign arbitrary binary data and include this signed data in an XML document. The result of encrypting data is an XML Signature element that contains or references the signature data.

The IBM provider implementation of Java XML Digital Signature supports these algorithms:

- Digest: SHA1
- Mac: HMAC-SHA1
- Signature: DSAwithSHA1, RSAwithSHA1
- Transformation: Canonicalization, Base64, XPath, and XSLT

History of changes

Version 7
: XML Digital Signatures now supports a Signature Engine for ECDSA signatures.

Version 6
: Java XML Encryption and Java XML Digital Signature are new for Version 6.

Documentation

The user guide and Javadoc information are on the developerWorks Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html

IBM SecureRandom Provider

IBM SecureRandom provides cryptographically strong random number generation as an alternative to the IBM JCE SecureRandom provider.

The IBM SecureRandom provider is available from Java 6 SR9 onwards. The provider contains implementations of MD5, SHA1, IBMSecureRandom, and SHA1PRNG. The algorithms MD5 and SHA1 are present only to supplement the SecureRandom implementations. The provider does not expose these algorithms as Message Digests that the provider supports.

The algorithms provided include:

Signature algorithms
: SHA1 with RSA

  MD5 with RSA

Random number generation algorithms
: IBMSecureRandom

  IBM SHA1PRNG

Differences between IBM SecureRandom and IBM JCE SecureRandom

The IBM SecureRandom algorithm in the IBM SecureRandom provider does not meet all of the government security specifications that are required for IBM JCE SecureRandom. As a consequence, the performance is improved. However, you

must use the IBM SecureRandom provider only for operations that do not need to meet the strict JCE specification. For example, the IBM SecureRandom provider can be used to replace IBM JCE SecureRandom when WebSphere Application Server 7.0 accesses an Oracle database server.

The IBM SecureRandom provider is intended only as a replacement for the SecureRandom algorithms of the IBMJCE provider. You must not use the IBM SecureRandom provider to replace the SecureRandom algorithms in any other provider, including IBMJCEFIPS.

How to implement the IBM SecureRandom provider

The `IBMSecureRandom.jar` file containing the provider can be found in the `jre/lib/ext` directory. The IBM SecureRandom provider is not in the provider list by default. If you want to use this provider, follow these steps:

1. Edit the following file: `jre/lib/security/java.security`.
2. Insert the IBM SecureRandom provider into the list before the IBM JCE provider. For example:

   ```
   ....
   security.provider.2=com.ibm.securerandom.provider.IBMSecureRandom
   security.provider.3=com.ibm.crypto.provider.IBMJCE
   ....
   ```
3. Save the file.

How to access the provider

The following extract shows you how to access the provider in your code:

```
        SecureRandom random=null;
        SecureRandom random1=null;

      try {
          random = SecureRandom.getInstance
("IBMSecureRandom","IBMSecureRandom");
          random1 = SecureRandom.getInstance("SHA1PRNG","IBMSecureRandom");

      } catch (NoSuchAlgorithmException nsae ) {
          System.out.println("No such alg : "+nsae);
          System.exit(-1);
      } catch (NoSuchProviderException nspe ) {

          System.out.println("No such provider : " + nspe);
          System.exit(-1);
      }
```

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:

- “First steps in problem determination” on page 212
- “AIX problem determination” on page 213
- “ORB problem determination” on page 246

© Copyright IBM Corp. 2011

- "NLS problem determination" on page 259

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 211 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 364.)

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 271 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 "Messages" on page 482.

AIX problem determination

This section describes problem determination on AIX.

The topics are:

- "Setting up and checking your AIX environment"
- "General debugging techniques" on page 216
- "Diagnosing crashes" on page 227
- "Debugging hangs" on page 228
- "Understanding memory usage" on page 231
- "Debugging performance problems" on page 239
- "MustGather information for AIX" on page 246

Setting up and checking your AIX environment

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

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

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

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

   The default installation directory is in `/usr/java7` for the 32-bit JVM and or /usr/java7_64 for the 64-bit JVM. For developer kits packaged with other products, the installation directory might be different; consult your product documentation.
2. Ensure that the **PATH** environment variable points to the correct Java executable (using `which java`), or that the application you are using is pointing to the correct Java directory. You must include /usr/java7/jre/bin:/usr/java7/bin in your **PATH** environment variable . If it is not present, add it by using `export PATH=/usr/java7/jre/bin:/usr/java7/bin:$PATH`.
3. Ensure that the **LANG** environment variable is set to a supported locale. You can find the language environment in use using `echo $LANG`, which should report one of the supported locales as documented in the *User Guide* shipped with the SDK.
4. Ensure that all the prerequisite AIX maintenance and APARs have been installed. The prerequisite APARs and filesets will have been checked during an install using `smitty` or `installp`. You can find the list of prerequisites in the *User Guide* that is shipped with the SDK. Use `lslpp -l` to find the list of current filesets. Use `instfix -i -k <apar number>` to test for the presence of an APAR and `instfix -i | grep _ML` to find the installed maintenance level.

Directory requirements

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

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

The system dump agents can also be configured on the command line. See “Using dump agents” on page 271 for more information.

Enabling full AIX core files:

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

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

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

Operating system settings

1. To obtain full system dumps, set the following `ulimit` options:

```
ulimit -c unlimited   turn on corefiles with unlimited size
ulimit -n unlimited   allows an unlimited number of open file descriptors
ulimit -d unlimited   sets the user data limit to unlimited
ulimit -f unlimited   sets the file limit to unlimited
```

You can display the current `ulimit` settings with:

```
ulimit -a
```

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

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

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

   Alternatively, you can run:

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

Java Virtual Machine settings

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

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

```
java -Xdump:what
```

which should include something like the following:

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

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

You can change and set options using the command-line option **`-Xdump`**, which is described in "Using dump agents" on page 271.

Available disk space

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

General debugging techniques

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

In addition to this information, you can obtain AIX publications from the IBM System p and AIX Information Center: http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp. Of particular interest are:

- Performance management and tuning
- Programming for AIX

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

There are several diagnostic tools available with the JVM to help diagnose problems:

- Starting Javadumps, see "Using Javadump" on page 286.
- Starting Heapdumps, see "Using Heapdump" on page 306.
- Starting system dumps, see "Using system dumps and the dump viewer" on page 315.

AIX provides various commands and tools that can be useful in diagnosing problems.

AIX debugging commands:

List of debugging commands.

bindprocessor –q
: Lists the available processors.

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

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

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

  The Java 5.0 SDK also includes a dbx Plug-in for additional help debugging Java applications. See "DBX Plug-in" on page 226 for more information.

iostat

: Reports the read and write rate to all disks. This tool can help determine if disk workload should be spread across multiple disks. **iostat** also reports the same CPU activity that **vmstat** does.

lsattr
: Details characteristics and values for devices in the system.

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

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

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

lsconf
Shows basic hardware and configuration details. See “lsconf” on page 218 for an example.

netpmon
uses the **trace** facility to obtain a detailed picture of network activity during a time interval. See “netpmon” on page 219 for an example.

netstat
Shows information about socket and network memory usage. Use this command with the **–m** option to look at mbuf memory usage. See “netstat” on page 220 for more details.

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

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

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

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

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

ps Shows process information. See “ps” on page 221 for more details.

sar
Shows usage by multiple CPUs. See “**sar**” on page 222 for more details.

svmon
Captures snapshots of virtual memory. See “svmon” on page 223 for more details.

tprof
The **tprof** command reports CPU usage for individual programs and the system as a whole. The command is useful for analyzing a Java program that might be CPU-bound. You can determine which sections of the program are most heavily using the CPU.

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

topas
A graphical interface to system activity. See “topas” on page 224 for more details.

trace
Captures a sequential flow of time-stamped system events. The trace is a valuable tool for observing system and application execution. See “trace” on page 225 for more details.

truss
Traces the following details for a process: system calls, dynamically loaded user-level function calls, received signals, and incurred machine faults.

vmstat
Reports statistics about kernel threads in the run and wait queue, memory paging, interrupts, system calls, context switches, and CPU activity. See "vmstat" on page 225 for more details.

lsconf:

This command shows basic hardware and configuration details.

For example:

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

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

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

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

INSTALLED RESOURCE LIST

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

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

+ sys0                            System Object
+ sysplanar0                      System Planar
* vio0                             Virtual I/O Bus
* vsa0                             LPAR Virtual Serial Adapter
* vty0                             Asynchronous Terminal
* pci12            U1.5-P2         PCI Bus
* pci11            U1.5-P2         PCI Bus
```

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

netpmon:

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

It also displays process CPU statistics that show:

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

For example,

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

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

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

The adapter usage is shown here:

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

The following example shows the java extract:

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

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

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

The following extract shows the thread output:

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

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

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

netstat:

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

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

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

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

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

-- At the end of the file:
```

```
Streams mblk statistic failures:
0 high priority mblk failures
0 medium priority mblk failures
0 low priority mblk failures
```

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

ps:

Shows process information.

The Process Status (ps) is used to monitor:

- A process.
- Whether the process is still consuming CPU cycles.
- Which threads of a process are still running.

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

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

Your output should be:

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

Where

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

PPID
: The Parent Process ID.

PID
: The Process ID.

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

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

TTY
: The controlling workstation for the process.

TIME
: The total execution time for the process.

CMD
: The full command name and its parameters.

To see which threads are still running, type:

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

Your output should be:

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

Where

USER
The user name of the person running the process.

TID
The Kernel Thread ID of each thread.

ST The state of the thread:

- **O** Nonexistent.
- **R** Running.
- **S** Sleeping.
- **W** Swapped.
- **Z** Canceled.
- **T** Stopped.

CP CPU utilization of the thread.

PRI
Priority of the thread.

SC Suspend count.

ARCHON
Wait channel.

F Flags.

TAT
Controlling terminal.

BAND
CPU to which thread is bound.

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

sar:

Use the **sar** command to check the balance of processor usage for multiple processors.

In this following example, two samples are taken every 5 seconds on a twin-processor system that is running at 80% utilization.

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

AIX aix4prt 0 5 000544144C00    02/09/01

15:29:32 cpu    %usr    %sys    %wio   %idle
15:29:37  0       34      46       0      20
```

```
          1       32      47       0      21
          -       33      47       0      20
15:29:42  0       31      48       0      21
          1       35      42       0      22
          -       33      45       0      22

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

svmon:

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

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

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

Gives output like:

```
     Pid Command        Inuse      Pin     Pgsp  Virtual   64-bit    Mthrd
   25084 AppS           78907     1570      182    67840        N        Y
  Vsid     Esid Type Description           Inuse   Pin Pgsp Virtual Addr Range
 2c7ea        3 work shmat/mmap            36678     0    0 36656   0..65513
 3c80e        4 work shmat/mmap             7956     0    0  7956   0..65515
 5cd36        5 work shmat/mmap             7946     0    0  7946   0..65517
 14e04        6 work shmat/mmap             7151     0    0  7151   0..65519
 7001c        d work shared library text    6781     0    0   736   0..65535
     0        0 work kernel seg             4218  1552  182  3602   0..22017 :
                                                                      65474..65535
 6cb5a        7 work shmat/mmap             2157     0    0  2157   0..65461
 48733        c work shmat/mmap             1244     0    0  1244   0..1243
  cac3        - pers /dev/hd2:176297        1159     0    -     -   0..1158
 54bb5        - pers /dev/hd2:176307         473     0    -     -   0..472
 78b9e        - pers /dev/hd2:176301         454     0    -     -   0..453
 58bb6        - pers /dev/hd2:176308         254     0    -     -   0..253
  cee2        - work                         246    17    0   246   0..49746
 4cbb3        - pers /dev/hd2:176305         226     0    -     -   0..225
 7881e        - pers /dev/e2axa702-1:2048    186     0    -     -   0..1856
 68f5b        - pers /dev/e2axa702-1:2048    185     0    -     -   0..1847
 28b8a        - pers /dev/hd2:176299         119     0    -     -   0..118
 108c4        - pers /dev/e2axa702-1:1843    109     0    -     -   0..1087
 24b68        f work shared library data      97     0    0    78   0..1470
 64bb9        - pers /dev/hd2:176311          93     0    -     -   0..92
 74bbd        - pers /dev/hd2:176315          68     0    -     -   0..67
 3082d        2 work process private          68     1    0    68   65287..65535
 10bc4        - pers /dev/hd2:176322          63     0    -     -   0..62
 50815        1 pers code,/dev/hd2:210969      9     0    -     -   0..8
 44bb1        - pers /dev/hd2:176303           7     0    -     -   0..6
 7c83e        - pers /dev/e2axa702-1:2048      4     0    -     -   0..300
 34a6c        a mmap mapped to sid 44ab0       0     0    -     -
 70b3d        8 mmap mapped to sid 1c866       0     0    -     -
 5cb36        b mmap mapped to sid 7cb5e       0     0    -     -
 58b37        9 mmap mapped to sid 1cb66       0     0    -     -
 1c7c7        - pers /dev/hd2:243801           0     0    -     -
```

in which:

Vsid
Segment ID

Esid
Segment ID: corresponds to virtual memory segment. The Esid maps to the

Virtual Memory Manager segments. By understanding the memory model that is being used by the JVM, you can use these values to determine whether you are allocating or committing memory on the native or Java heap.

Type
Identifies the type of the segment:

pers Indicates a persistent segment.

work Indicates a working segment.

clnt Indicates a client segment.

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

Description

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

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

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

kernel
The segment is used by the kernel.

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

process private
Private data for the process.

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

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

Pin
The number of pages pinned from this segment.

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

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

topas:

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

The screen looks like this:

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

trace:

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

While many of the other tools provide general statistics such as CPU and I/O utilization, the trace facility provides more detailed information. For example, you can find out:

- Where an event occurred in the code.
- Which process caused an event to occur.
- When an event took place.
- How an event is affecting the system.

The **curt** postprocessing tool can extract information from the trace. It provides statistics on CPU utilization and process and thread activity. Another postprocessing tool is **splat**, the Simple Performance Lock Analysis Tool. This tool can be used to analyze simple locks in the AIX kernel and kernel extensions.

vmstat:

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

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

The general syntax of this command is:

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

A typical output looks like this:

```
kthr     memory             page              faults        cpu        time
----- ----------- ------------------------ ------------ ----------- --------
 r  b   avm   fre  re  pi  po  fr   sr  cy  in   sy  cs us sy id wa hr mi se
```

```
 0  0 45483   221   0   0   0   0    1   0 224  326 362 24  7 69  0 15:10:22
 0  0 45483   220   0   0   0   0    0   0 159   83  53  1  1 98  0 15:10:23
 2  0 45483   220   0   0   0   0    0   0 145  115  46  0  9 90  1 15:10:24
```

In this output, look for:

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

DBX Plug-in:

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

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

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

```
pluginload /usr/java7/jre/bin/libdbx_j9.so
```

You can also set the **DBX_PLUGIN_PATH** environment variable to /usr/java7/jre/bin . DBX automatically loads any Plug-ins found in the path given.

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

```
plugin java help
```

from the DBX prompt.

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

```
dbx /usr/java7/jre/bin/java
```

Under DBX, issue the command:

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

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

```
dbx -c .dbxinit
```

where `.dbxinit` is the default command file.

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

Diagnosing crashes

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

Key data files:

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

- To find the core file, look for the location shown in the JVMDUMP messages issued by the JVM.
- Process the core file using the **jextract** utility. For more information, see "Using jextract" on page 318.
- Collect the Javadump file. To find the Javadump file, look for the location shown in the JVMDUMP messages issued by the JVM.
- Collect any stdout and stderr output generated by the Java process
- Collect the system error report:

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

These steps generate the following files:

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

Locating the point of failure:

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

If **dbx** produces no stack trace, the crash might have two possible causes:

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

A failing instruction reported by **dbx** as `stwu` indicates that there might have been a stack overflow. For example:

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

You can check for the first cause by using the **dbx** command `thread info` and looking at the stack pointer, stack limit, and stack base values for the current thread. If the value of the stack pointer is close to that of the stack base, you might have had a stack overflow. A stack overflow occurs because the stack on AIX grows from the stack limit downwards, towards the stack base. If the problem is a native stack overflow, you can solve the overflow by increasing the size of the native stack from the default size of 400K using the command-line option

-Xss<size>. Check for a stack overflow, regardless of the failing instruction. To reduce the possibility of a JVM crash, you must set an appropriate native stack size when you run a Java program using a lot of native stack.

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


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

For the second cause, **dbx** does not understand the structure of the JIT and Interpreter stack frames, and is not capable of generating a stack trace from them. The Javadump, however, does not suffer from this limitation and can be used to examine the stack trace. A failure in JIT-compiled code can be verified and examined using the JIT problem determination guidance, see "JIT and AOT problem determination" on page 364.

Debugging hangs

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

This lack of response can be caused because:

- The process has come to a complete halt because of a deadlock condition
- The process has become caught in an infinite loop
- The process is running very slowly

AIX deadlocks:

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

The **ps** command is described in "AIX debugging commands" on page 216. For example:

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

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

For an explanation of deadlocks and how the Javadump tool is used to diagnose them, see "Locks, monitors, and deadlocks (LOCKS)" on page 295.

AIX busy hangs:

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

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

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

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

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

To see how the javadump tool is used to diagnose loops, see "Threads and stack trace (THREADS)" on page 296. If you cannot diagnose the problem from the javadump and if the process still seems to be using processor cycles, either it has entered an infinite loop or it is suffering from very bad performance. Using **ps -mp [process id] -o THREAD** allows individual threads in a particular process to be monitored to determine which threads are using the CPU time. If the process has entered an infinite loop, it is likely that a small number of threads will be using the time. For example:

```
$ ps -mp 43824 -o THREAD
    USER   PID  PPID     TID ST  CP PRI SC    WCHAN        F     TT BND COMMAND
  wsuser 43824 51762       - A   66  60 77        *   200001  pts/4   - java ...
       -     -     -    4021 S    0  60  1 22c4d670   c00400      -   - -
       -     -     -   11343 S    0  60  1 e6002cbc  8400400      -   - -
       -     -     -   14289 S    0  60  1 22c4d670   c00400      -   - -
       -     -     -   14379 S    0  60  1 22c4d670   c00400      -   - -
...
       -     -     -   43187 S    0  60  1 701e6114   400400      -   - -
       -     -     -   43939 R   33  76  1 20039c88   c00000      -   - -
       -     -     -   50275 S    0  60  1 22c4d670   c00400      -   - -
       -     -     -   52477 S    0  60  1 e600ccbc  8400400      -   - -
...
       -     -     -   98911 S    0  60  1 7023d46c   400400      -   - -
       -     -     -   99345 R   33  76  0        -   400000      -   - -
       -     -     -   99877 S    0  60  1 22c4d670   c00400      -   - -
       -     -     -  100661 S    0  60  1 22c4d670   c00400      -   - -
       -     -     -  102599 S    0  60  1 22c4d670   c00400      -   - -
...
```

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

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

```
 $t61    wait                 running    18791     k   no   sys  getLinkRegister
 $t62    wait                 running    99345     k   no   sys  getLinkRegister
 $t63    wait                 running    20995     k   no   sys  getLinkRegister
```

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

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

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

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

```
"Worker#31" (TID:0x36288b10, sys_thread_t:0x220c2db8) Native Thread State:
ThreadID: 00003233 Reuse: 1 USER SUSPENDED Native Stack Data : base: 22107f80
pointer 22106390 used(7152) free(250896)
----- Monitors held -----
java.io.OutputStreamWriter@3636a930
com.ibm.servlet.engine.webapp.BufferedWriter@3636be78
com.ibm.servlet.engine.webapp.WebAppRequestDispatcher@3636c270
com.ibm.servlet.engine.srt.SRTOutputStream@36941820
com.ibm.servlet.engine.oselistener.nativeEntry.NativeServerConnection@36d84490 JNI pinning lock

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

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

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

com.ibm.servlet.engine.oselistener.nativeEntry.NativeServerConnection.write(Compiled Code)
com.ibm.servlet.engine.srp.SRPConnection.write(Compiled Code)
com.ibm.servlet.engine.srt.SRTOutputStream.write(Compiled Code)
java.io.OutputStreamWriter.flushBuffer(Compiled Code)
java.io.OutputStreamWriter.flush(Compiled Code)
java.io.PrintWriter.flush(Compiled Code)
```

```
com.ibm.servlet.engine.webapp.BufferedWriter.flushChars(Compiled Code)
com.ibm.servlet.engine.webapp.BufferedWriter.write(Compiled Code)
java.io.Writer.write(Compiled Code)
java.io.PrintWriter.write(Compiled Code)
java.io.PrintWriter.write(Compiled Code)
java.io.PrintWriter.print(Compiled Code)
java.io.PrintWriter.println(Compiled Code)
pagecompile._identifycustomer_xjsp.service(Compiled Code)
javax.servlet.http.HttpServlet.service(Compiled Code)
com.ibm.servlet.jsp.http.pagecompile.JSPState.service(Compiled Code)
com.ibm.servlet.jsp.http.pagecompile.PageCompileServlet.doService(Compiled Code)
com.ibm.servlet.jsp.http.pagecompile.PageCompileServlet.doGet(Compiled Code)
javax.servlet.http.HttpServlet.service(Compiled Code)
javax.servlet.http.HttpServlet.service(Compiled Code)
```

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

Poor performance on AIX:

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

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

See “Debugging performance problems” on page 239 for more information about performance on AIX.

Understanding memory usage

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

32- and 64-bit JVMs:

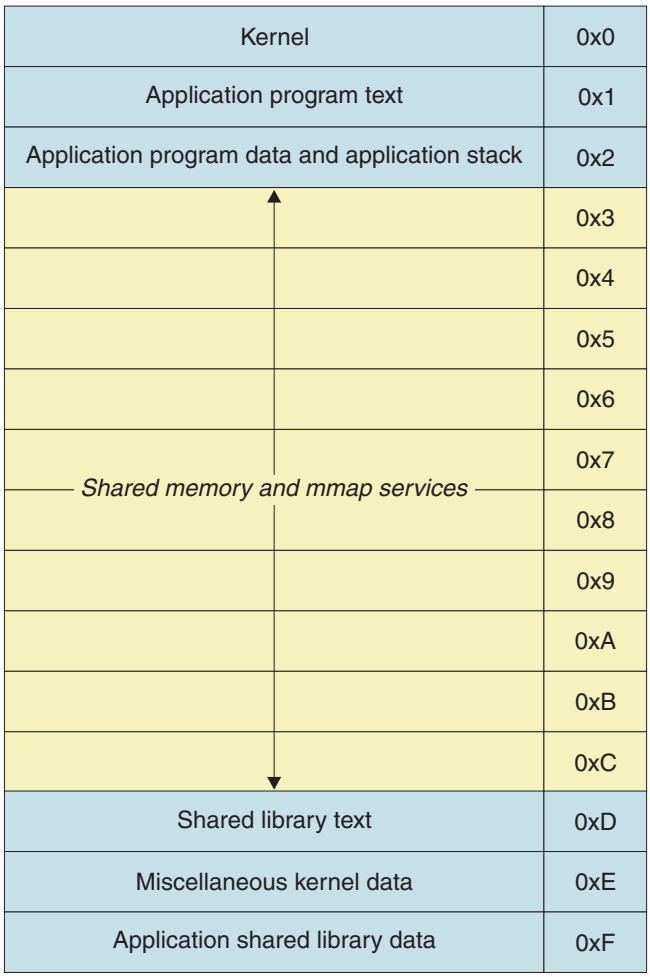
Most of the information in this section about altering the memory model and running out of native heap is relevant only to the 32-bit model, because the 64-bit model does not suffer from the same kind of memory constraints.

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

The 32-bit AIX Virtual Memory Model:

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

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



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

The 64-bit AIX Virtual Memory Model:

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

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

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

Changing the Memory Model (32-bit JVM):

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

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

The small memory model

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

The large memory model

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

You can alter the *o_maxdata* setting by:

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

The very large memory model

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

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

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

The native and Java heaps:

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

The Java heap contains the instances of Java objects and is often referred to as 'the heap'. It is the Java heap that is maintained by Garbage Collection, and it is the Java heap that is changed by the command-line heap settings. The Java heap is

allocated using mmap, or shmat if large page support is requested. The maximum size of the Java heap is preallocated during JVM startup as one contiguous area, even if the minimum heap size setting is lower. This allocation allows the artificial heap size limit imposed by the minimum heap size setting to move toward the actual heap size limit with heap expansion.

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

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

The AIX 32-bit JVM default memory models: The AIX 5.0 Java launcher alters its **MAXDATA** setting in response to the command-line options to optimize the amount of memory available to the process. The default are as follows:

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

Monitoring the native heap:

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

Use svmon like this:

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

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

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

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

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

    Vsid      Esid Type Description              LPage  Inuse   Pin Pgsp Virtual
    50e9         - work                             -  41382     0    0 41382
                   Addr Range: 0..41381
    9dfb         - work                             -  28170     0 2550 30720
                   Addr Range: 0..30719
    ddf3         3 work working storage             -   9165     0  979 10140
                   Addr Range: 0..16944
       0         0 work kernel seg                  -   5118  4766 1322  6420
                   Addr Range: 0..11167
```

```
c819         d work text or shared-lib code seg  -   2038     0  283  6813
               Addr Range: 0..10219
2ded         f work working storage              -    191     0   20   224
               Addr Range: 0..4150
f5f6         - work                              -     41    14    4    45
               Addr Range: 0..49377
6e05         2 work process private              -     35     2   23    58
               Addr Range: 65296..65535
1140         6 work other segments               -     26     0    0    26
               Addr Range: 0..32780
cdf1         - work                              -      2     0    0     2
               Addr Range: 0..5277
e93f         - work                              -      0     0    0     0
3164         c mmap mapped to sid 1941           -      0     0    -     -
2166         - work                              -      0     0    0     0
496b         b mmap mapped to sid 2166           -      0     0    -     -
b51e         - clnt /dev/fslv00:44722            -      0     0    -     -
               Addr Range: 0..207
ee1c         a mmap mapped to sid e93f           -      0     0    -     -
1941         - work                              -      0     0    0     0
1081         7 mmap mapped to sid 9dfb           -      0     0    -     -
edf5         8 mmap mapped to sid 50e9           -      0     0    -     -
c01b         9 mmap mapped to sid cdf1           -      0     0    -     -
```

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

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

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

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

Native heap usage:

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

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

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

```
(warm) Compiling java/lang/System.getEncoding(I)Ljava/lang/String;
+ (warm) java/lang/System.getEncoding(I)Ljava/lang/String; @ 0x02BA0028-0x02BA0113
                                        (2) Compiling java/lang/String.hashCode()I
+ (warm) java/lang/String.hashCode()I @ 0x02BA0150-0x02BA0229
  (2) Compiling java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object;)
   Ljava/lang/Object;
+ (warm) java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object;)
                 Ljava/lang/Object; @ 0x02BA0270-0x02BA03F7
                (2) Compiling java/lang/String.charAt(I)C
+ (warm) java/lang/String.charAt(I)C @ 0x02BA0430-0x02BA04AC
       (2) Compiling java/util/Locale.toLowerCase(Ljava/lang/String;)
                Ljava/lang/String;
+ (warm) java/util/Locale.toLowerCase(Ljava/lang/String;)Ljava/lang/String;
                                                   @ 0x02BA04D0-0x02BA064C
```

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

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

Specifying MALLOCTYPE:

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

For more information, see: System Memory Allocation Using the malloc Subsystem.

Monitoring the Java heap:

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

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

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

where:

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

See “Garbage Collector diagnostics” on page 375 for more information about verbose GC output and monitoring.

Receiving OutOfMemoryError exceptions:

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

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

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

The maximum size of an object that can be allocated is limited only by available memory. The maximum number of array elements supported is 2^31 - 1, the maximum permitted by the Java Virtual Machine specification. In practice, you

might not be able to allocate large arrays due to available memory. Configure the total amount of memory available for objects using the **-Xmx** command-line option.

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

Is the Java or native heap exhausted?:

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

If a received OutOfMemory condition has one of these codes or messages, consulting “Messages” on page 482 might point to the origin of the error, either native or Java heap.

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

Java heap exhaustion:

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

Garbage collection can free only objects that are no longer referenced by other objects, or are referenced from the thread stacks (see “Memory management” on page 23 for more details).

Java heap exhaustion can be identified from the **-verbose:gc** output by garbage collection occurring more and more frequently, with less memory being freed. Eventually the JVM will fail, and the heap occupancy will be at, or almost at, 100% (See “Memory management” on page 23 for more details on **-verbose:gc** output).

If the Java heap is being exhausted, and increasing the Java heap size does not solve the problem, the next stage is to examine the objects that are on the heap, and look for suspect data structures that are referencing large numbers of Java objects that should have been released. Use Heapdump Analysis, as detailed in “Using Heapdump” on page 306. Similar information can be gained by using other tools, such as JProbe and OptimizeIt.

Native heap exhaustion:

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

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

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

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

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

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

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

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

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

MALLOCMULTIHEAP=heaps:n,considersize
By default, the malloc subsystem uses a single heap. **MALLOCMULTIHEAP** lets users enable the use of multiple heaps of memory. Multiple heaps of memory can lead to memory fragmentation, and so the use of this environment variable is to be avoided.

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

Note: These options might cause a percentage of performance impact. Also the 3.1 malloc allocator does not support the `Malloc Multiheap` and `Malloc Buckets` options.

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

Submitting a bug report:

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

The process for raising a bug is detailed in “Submitting problem reports” on page 211, and the data that should be included in the bug report is listed as follows:

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

Debugging performance problems

Locating the causes of poor performance is often difficult. Although many factors can affect performance, the overall effect is generally perceived as poor response or slow execution of your program.

Correcting one performance problem might cause more problems in another area. By finding and correcting a bottleneck in one place you might only shift the cause of poor performance to other areas. To improve performance, experiment with tuning different parameters, monitoring the effect, and retuning until you are satisfied that your system is performing acceptably

Finding the bottleneck:

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

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

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

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

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

```
> vmstat 1 10
```

outputs:

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

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

The example above shows a system that is CPU bound. This can be seen as the user (us) plus system (sy) CPU values either equal or are approaching 100. A system that is memory bound shows significant values of page in (pi) and page out (po). A system that is disk I/O bound will show an I/O wait percentage (wa) exceeding 10%. More details of vmstat can be found in "AIX debugging commands" on page 216.

CPU bottlenecks:

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

The recommended tool is **tprof**:

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

outputs:

```
Mon Nov 28 12:40:11 2010
System: AIX 6.1 Node: voodoo Machine: 00455F1B4C00

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

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

```
=======                   ===      ===  ===== ======   ==== ======  =====
Total                                   100.00  64.70   0.00  35.29   0.00

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

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

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

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

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

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

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

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

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

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

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

```
Shared Object                                                  %
=============                                                          ======
/work/JDK/inst.images/rios_aix32_7/sdk/jre/bin/libj9gc26.so  17.42
/usr/lib/libc.a[shr.o]                                                   9.38
/usr/lib/libpthreads.a[shr_xpg5.o]                                       6.94
/work/JDK/inst.images/rios_aix32_7/sdk/jre/bin/libj9thr26.so   1.03
/work/JDK/inst.images/rios_aix32_7/sdk/jre/bin/libj9prt26.so   0.24
/work/JDK/inst.images/rios_aix32_7/sdk/jre/bin/libj9vm26.so   0.10
/work/JDK/inst.images/rios_aix32_7/sdk/jre/bin/libj9ute26.so   0.06
/work/JDK/inst.images/rios_aix32_7/sdk/jre/bin/libj9jit26.so   0.05
/usr/lib/libtrace.a[shr.o]                                               0.04
/work/JDK/inst.images/rios_aix32_7/sdk/jre/bin/libj9trc26.so   0.02
/work/JDK/inst.images/rios_aix32_7/sdk/jre/bin/libj9hookable26.so   0.01
```

This section shows that almost all of the time is being spent in one particular shared library, which is part of the JVM installation: `libj9gc26.so`. By

understanding the functions that the more commonly used JVM libraries carry out, it becomes possible to build a more accurate picture of what the threads are doing:

libbcv26.so
Bytecode Verifier

libdbg26.so
Debug Server (used by the Java Debug Interface)

libj9gc26.so
Garbage Collection

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

libj9jit26.so
The Just In Time (JIT) Compiler

libj9jvmti26.so
The JVMTI interface

libj9prt26.so
The "port layer" between the JVM and the Operating System

libj9shr26.so
The shared classes library

libj9thr26.so
The threading library

libj9ute26.so
The trace engine

libj9vm26.so
The core Virtual Machine

libj9zlib26.so
The compressed file utility library

libjclscar_26.so
The Java Class Library (JCL) support routines

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

Again, you can obtain a more accurate understanding of what is occurring inside the `libj9gc26.so` library during the CPU time by analyzing the relevant section of the **tprof** output:

```
Profile: /work/JDK/inst.images/rios_aix32_7/sdk/jre/bin/
                                                  libj9gc26.so

  Total % For All Processes (/work/JDK/inst.images/rios_aix32_7/
                                sdk/jre/bin/libj9gc26.so) = 17.42

Subroutine                                            %   Source
==========                                         ====== ======
Scheme::scanMixedObject(MM_Environment*,J9Object*)   2.67 MarkingScheme.cpp
MarkingScheme::scanClass(MM_Environment*,J9Class*)   2.54 MarkingScheme.cpp
.GC_ConstantPoolObjectSlotIterator::nextSlot()       1.96 jectSlotIterator.cpp
lelTask::handleNextWorkUnit(MM_EnvironmentModron*)   1.05 ParallelTask.cpp
orkPackets::getPacket(MM_Environment*,MM_Packet**)   0.70 WorkPackets.cpp
cheme::fixupRegion(J9Object*,J9Object*,bool,long&)   0.67 CompactScheme.cpp
WorkPackets::putPacket(MM_Environment*,MM_Packet*)   0.47 WorkPackets.cpp
rkingScheme::scanObject(MM_Environment*,J9Object*)   0.43 MarkingScheme.cpp
```

```
sweepChunk(MM_Environment*,MM_ParallelSweepChunk*)   0.42 allelSweepScheme.cpp
ment*,J9IndexableObject*,J9Object**,unsigned long)   0.38 MarkingScheme.cpp
M_CompactScheme::getForwardingPtr(J9Object*) const   0.36 CompactScheme.cpp
ObjectHeapIteratorAddressOrderedList::nextObject()   0.33 dressOrderedList.cpp
ckets::getInputPacketFromOverflow(MM_Environment*)   0.32 WorkPackets.cpp
.MM_WorkStack::popNoWait(MM_Environment*)            0.31 WorkStack.cpp
WorkPackets::getInputPacketNoWait(MM_Environment*)   0.29 WorkPackets.cpp
canReferenceMixedObject(MM_Environment*,J9Object*)   0.29 MarkingScheme.cpp
MarkingScheme::markClass(MM_Environment*,J9Class*)   0.27 MarkingScheme.cpp
._ptrgl                                              0.26 ptrgl.s
_MarkingScheme::initializeMarkMap(MM_Environment*)   0.25 MarkingScheme.cpp
.MM_HeapVirtualMemory::getHeapBase()                 0.23 eapVirtualMemory.cpp
```

This output shows that the most-used functions are:

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

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

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

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

For the previous example:

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

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

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

These entries show that, in this case, the thread is calling GC, and explains the time spent in the `libj9gc26.so` shared library.

Memory bottlenecks:

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

Use the **svmon** tool:

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

This command outputs:

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

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

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

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

I/O bottlenecks:

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

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

JVM heap sizing:

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

See “How to do heap sizing” on page 45 for information on how to correctly set the size of your heap.

JIT compilation and performance:

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

The JIT is another area that can affect the performance of your program. The performance of short-running applications can be improved by using the **-Xquickstart** command-line parameter. The JIT is switched on by default, but you can use **-Xint** to turn it off. You also have considerable flexibility in controlling JIT processing. For more details about the JIT, see "The JIT compiler" on page 56 and "JIT and AOT problem determination" on page 364.

Application profiling:

You can learn a lot about your Java application by using the hprof profiling agent

Statistics about CPU and memory usage are presented along with many other options. The hprof tool is discussed in detail in "Using the HPROF Profiler" on page 425. **-Xrunhprof:help** gives you a list of suboptions that you can use with hprof.

Testing JVM optimizations:

Performance problems might be associated with new optimizations that have been introduced for IBM SDK for AIX, v7.

Java monitor optimizations

IBM SDK for AIX, v7 introduces new optimizations that are expected to improve CPU efficiency. However, there might be some situations where lower CPU utilization is achieved, but overall application performance decreases. You can test whether the new optimizations are negatively affecting your application by reverting to the behavior of earlier versions.

- If performance is affected 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

IBM SDK for AIX, v7 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:

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

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

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

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

MustGather information for AIX

The information that is most useful at a point of failure depends, in general, on the type of failure that is experienced.

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 has occurred. 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 370.

The AIX core file

If the environment is correctly set up to produce full AIX Core files (as detailed in "Setting up and checking your AIX environment" on page 213), a core file is generated when the process receives a terminal signal (that is, SIGSEGV, SIGILL, or SIGABORT). The core file is generated into the current working directory of the process, or at the location pointed to by the label field specified with **-Xdump**.

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

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

When a core file is generated:

1. Run the **jextract** utility against the core file, see topic "Using jextract" on page 318. Running the **jextract** utility generates a file in the current directory called `dumpfilename.zip`. This compressed file contains the dump and the required Java executable file and libraries.
2. If **jextract** processing fails, use the snapcore utility to collect the same information. For example, `snapcore -d /tmp/savedir core.001 /usr/java5/jre/bin/java` creates an archive (snapcore_pid.pax.Z) in the file /tmp/savedir.

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

The javacore file:

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

The Error Report

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

ORB problem determination

One of your first tasks when debugging an ORB problem is to determine whether the problem is in the client-side or in the server-side of the distributed application.

Think of a typical RMI-IIOP session as a simple, synchronous communication between a client that is requesting access to an object, and a server that is providing it.

During this communication, a problem might occur in the execution of one of the following steps:

1. The client writes and sends a request to the server.
2. The server receives and reads the request.
3. The server executes the task in the request.
4. The server writes and sends a reply back.
5. The client receives and reads the reply.

It is not always easy to identify where the problem occurred. Often, the information that the application returns, in the form of stack traces or error messages, is not enough for you to make a decision. Also, because the client and server communicate through their ORBs, if a problem occurs, both sides will probably record an exception or unusual behavior.

This section describes all the clues that you can use to find the source of the ORB problem. It also describes a few common problems that occur more frequently. The topics are:

- "Identifying an ORB problem"
- "Debug properties" on page 248
- "ORB exceptions" on page 249
- "Interpreting the stack trace" on page 252
- "Interpreting ORB traces" on page 253
- "Common problems" on page 256
- "IBM ORB service: collecting data" on page 258

Identifying an ORB problem

A background of the constituents of the IBM ORB component.

What the ORB component contains

The ORB component contains the following:

- Java ORB from IBM and rmi-iiop runtime (com.ibm.rmi.*, com.ibm.CORBA.*)
- RMI-IIOP API (javax.rmi.CORBA.*,org.omg.CORBA.*)
- IDL to Java implementation (org.omg.* and IBM versions com.ibm.org.omg.*)
- Transient name server (com.ibm.CosNaming.*, org.omg.CosNaming.*) - tnameserv
- -iiop and -idl generators (com.ibm.tools.rmi.rmic.*) for the rmic compiler - rmic
- idlj compiler (com.ibm.idl.*)

What the ORB component does not contain

The ORB component does *not* contain:

- RMI-JRMP (also known as Standard RMI)
- JNDI and its plug-ins

Therefore, if the problem is in java.rmi.* or sun.rmi.*, it is not an ORB problem. Similarly, if the problem is in com.sun.jndi.*, it is not an ORB problem.

Platform dependent problems

If possible, run the test case on more than one platform. All the ORB code is shared. You can nearly always reproduce genuine ORB problems on any platform. If you have a platform-specific problem, it is likely to be in some other component.

JIT problem

JIT bugs are very difficult to find. They might show themselves as ORB problems. When you are debugging or testing an ORB application, it is always safer to switch off the JIT by setting the option **-Xint**.

Fragmentation

Disable fragmentation when you are debugging the ORB. Although fragmentation does not add complications to the ORB's functioning, a fragmentation bug can be difficult to detect because it will most likely show as a general marshalling problem. The way to disable fragmentation is to set the ORB property **com.ibm.CORBA.FragmentSize=0**. You must do this on the client side and on the server side.

ORB versions

The ORB component carries a few version properties that you can display by calling the main method of the following classes:

1. `com.ibm.CORBA.iiop.Version` (ORB runtime version)
2. com.ibm.tools.rmic.iiop.Version (for tools; for example, **idlj** and **rmic**)
3. rmic -iiop -version (run the command line for **rmic**)

Limitation with bidirectional GIOP

Bidirectional GIOP is not supported.

Debug properties

Properties to use to enable ORB traces.

Attention: Do not enable tracing for normal operation, because it might cause performance degradation. Even if you have switched off tracing, FFDC (First Failure Data Capture) is still working, so that only serious errors are reported. If a debug file is produced, examine it to check on the problem. For example, the server might have stopped without performing an ORB.shutdown().

You can use the following properties to enable the ORB traces:

- **com.ibm.CORBA.Debug:** This property turns on trace, message, or both. If you set this property to **trace**, only traces are enabled; if set to **message**, only messages are enabled. When set to **true**, both types are enabled; when set to **false**, both types are disabled. The default is **false**.
- **com.ibm.CORBA.Debug.Output:** This property redirects traces to a file, which is known as a trace log. When this property is not specified, or it is set to an empty string, the file name defaults to the format orbtrc.DDMMYYYY.HHmm.SS.txt, where D=Day; M=Month; Y=Year; H=Hour (24 hour format); m=Minutes; S=Seconds. If the application (or Applet) does not have the privilege that it requires to write to a file, the trace entries go to stderr.
- **com.ibm.CORBA.CommTrace:** This property turns on wire tracing, also known as Comm tracing. Every incoming and outgoing GIOP message is sent to the

trace log. You can set this property independently from Debug. This property is useful if you want to look only at the flow of information, and you are not interested in debugging the internals. The only two values that this property can have are **true** and **false**. The default is **false**.

Here is an example of common usage:

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

For rmic -iiop or rmic -idl, the following diagnostic tools are available:

- **-J-Djavac.dump.stack=1:** This tool ensures that all exceptions are caught.
- **-Xtrace:** This tool traces the progress of the parse step.

If you are working with an IBM SDK, you can obtain CommTrace for the transient name server (tnameserv) by using the standard environment variable **IBM_JAVA_OPTIONS**. In a separate command session to the server or client SDKs, you can use:

```
export IBM_JAVA_OPTIONS=-Dcom.ibm.CORBA.CommTrace=true -Dcom.ibm.CORBA.Debug=true
```

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 fatal exceptions and should always be handled by the application. Therefore, if you get one of these user exceptions, you know where the problem is, because the application developer had to make allowance for such an exception to occur. In most of these cases, the ORB is not the source of the problem.

System exceptions

System exceptions are thrown transparently to the application and represent an unusual condition in which the ORB cannot recover gracefully, such as when a connection is dropped. The CORBA 2.6 specification defines 31 system exceptions and their mapping to Java. They all belong to the org.omg.CORBA package. The CORBA specification defines the meaning of these exceptions and describes the conditions in which they are thrown.

The most common system exceptions are:

- **BAD_OPERATION:** This exception is thrown when an object reference denotes an existing object, but the object does not support the operation that was called.
- **BAD_PARAM:** This exception is thrown when a parameter that is passed to a call is out of range or otherwise considered not valid. An ORB might raise this exception if null values or null pointers are passed to an operation.
- **COMM_FAILURE:** This exception is raised if communication is lost while an operation is in progress, after the request was sent by the client, but before the reply from the server has been returned to the client.
- **DATA_CONVERSION:** This exception is raised if an ORB cannot convert the marshaled representation of data into its native representation, or cannot convert the native representation of data into its marshaled representation. For example, this exception can be raised if wide character codeset conversion fails, or if an ORB cannot convert floating point values between different representations.
- **MARSHAL:** This exception indicates that the request or reply from the network is structurally not valid. This error typically indicates a bug in either the client-side or server-side runtime. For example, if a reply from the server indicates that the message contains 1000 bytes, but the actual message is shorter or longer than 1000 bytes, the ORB raises this exception.
- **NO_IMPLEMENT:** This exception indicates that although the operation that was called exists (it has an IDL definition), no implementation exists for that operation.
- **UNKNOWN:** This exception is raised if an implementation throws a non-CORBA exception, such as an exception that is specific to the implementation's programming language. It is also raised if the server returns a system exception that is unknown to the client. If the server uses a later version of CORBA than the version that the client is using, and new system exceptions have been added to the later version this exception can happen.

Completion status and minor codes

Two pieces of data are associated with each system exception, these are described in this section.

- A completion status, which is an enumerated type that has three values: COMPLETED_YES, COMPLETED_NO and COMPLETED_MAYBE. These values indicate either that the operation was executed in full, that the operation was not executed, or that the execution state cannot be determined.
- A long integer, called minor code, that can be set to some ORB vendor-specific value. CORBA also specifies the value of many minor codes.

Usually the completion status is not very useful. However, the minor code can be essential when the stack trace is missing. In many cases, the minor code identifies the exact location of the ORB code where the exception is thrown and can be used by the vendor's service team to localize the problem quickly. However, for standard CORBA minor codes, this is not always possible. For example:

```
org.omg.CORBA.OBJECT_NOT_EXIST: SERVANT_NOT_FOUND   minor code: 4942FC11  completed: No
```

Minor codes are usually expressed in hexadecimal notation (except for 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 484 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
org.omg.CORBA.ORB	resolve_initial_references	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_is_a	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_non_existent	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	OutputStream _request (String, boolean)	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_get_interface_def	java.net.SocketPermission connect
org.omg.CORBA. Request	invoke	java.net.SocketPermission connect
org.omg.CORBA. Request	send_deferred	java.net.SocketPermission connect
org.omg.CORBA. Request	send_oneway	java.net.SocketPermission connect

Class/Interface	Method	Required permission
javax.rmi. PortableRemoteObject	narrow	java.net.SocketPermission connect

If your program uses any of these methods, ensure that it is granted the necessary permissions.

Interpreting the stack trace

Whether the ORB is part of a middleware application or you are using a Java stand-alone application (or even an applet), you must retrieve the stack trace that is generated at the moment of failure. It could be in a log file, or in your terminal or browser window, and it could consist of several chunks of stack traces.

The following example describes a stack trace that was generated by a server ORB running in the WebSphere Application Server:

```
org.omg.CORBA.MARSHAL: com.ibm.ws.pmi.server.DataDescriptor; IllegalAccessException  minor code: 4942F23E
completed: No
         at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:199)
         at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
         at com.ibm.rmi.io.ValueHandlerImpl.read_Array(ValueHandlerImpl.java:625)
         at com.ibm.rmi.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:273)
         at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:189)
         at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
         at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie._invoke(_EJSRemoteStatelessPmiService_
Tie.java:613)
         at com.ibm.CORBA.iiop.ExtendedServerDelegate.dispatch(ExtendedServerDelegate.java:515)
         at com.ibm.CORBA.iiop.ORB.process(ORB.java:2377)
         at com.ibm.CORBA.iiop.OrbWorker.run(OrbWorker.java:186)
         at com.ibm.ejs.oa.pool.ThreadPool$PooledWorker.run(ThreadPool.java:104)
         at com.ibm.ws.util.CachedThread.run(ThreadPool.java:137)
```

In the example, the ORB mapped a Java exception to a CORBA exception. This exception is sent back to the client later as part of a reply message. The client ORB reads this exception from the reply. It maps it to a Java exception (java.rmi.RemoteException according to the CORBA specification) and throws this new exception back to the client application.

Along this chain of events, often the original exception becomes hidden or lost, as does its stack trace. On early versions of the ORB (for example, 1.2.x, 1.3.0) the only way to get the original exception stack trace was to set some ORB debugging properties. Newer versions have built-in mechanisms by which all the nested stack traces are either recorded or copied around in a message string. When dealing with an old ORB release (1.3.0 and earlier), it is a good idea to test the problem on newer versions. Either the problem is not reproducible (known bug already solved) or the debugging information that you obtain is much more useful.

Description string:

The example stack trace shows that the application has caught a CORBA org.omg.CORBA.MARSHAL system exception. After the MARSHAL exception, some extra information is provided in the form of a string. This string should specify minor code, completion status, and other information that is related to the problem. Because CORBA system exceptions are alarm bells for an unusual condition, they also hide inside what the real exception was.

Usually, the type of the exception is written in the message string of the CORBA exception. The trace shows that the application was reading a value (read_value()) when an IllegalAccessException occurred that was associated to class com.ibm.ws.pmi.server.DataDescriptor. This information is an indication of the real problem and should be investigated first.

Interpreting ORB traces

The ORB trace file contains messages, trace points, and wire tracing. This section describes the various types of trace.

Message trace:

An example of a message trace.

Here is a simple example of a message:

```
19:12:36.306 com.ibm.rmi.util.Version logVersions:110 P=754534:O=0:CT
ORBRas[default] IBM Java ORB build orbdev-20050927
```

This message records the time, the package, and the method name that was called. In this case, logVersions() prints out, to the log file, the version of the running ORB.

After the first colon in the example message, the line number in the source code where that method invocation is done is written (110 in this case). Next follows the letter P that is associated with the process number that was running at that moment. This number is related (by a hash) to the time at which the ORB class was loaded in that process. It is unlikely that two different processes load their ORBs at the same time.

The following O=0 (alphabetic O = numeric 0) indicates that the current instance of the ORB is the first one (number 0). CT specifies that this is the main (control) thread. Other values are: LT for listener thread, RT for reader thread, and WT for worker thread.

The ORBRas field shows which RAS implementation the ORB is running. It is possible that when the ORB runs inside another application (such as a WebSphere application), the ORB RAS default code is replaced by an external implementation.

The remaining information is specific to the method that has been logged while executing. In this case, the method is a utility method that logs the version of the ORB.

This example of a possible message shows the logging of entry or exit point of methods, such as:

```
14:54:14.848 com.ibm.rmi.iiop.Connection <init>:504 LT=0:P=650241:O=0:port=1360 ORBRas[default] Entry
.....
14:54:14.857 com.ibm.rmi.iiop.Connection <init>:539 LT=0:P=650241:O=0:port=1360 ORBRas[default] Exit
```

In this case, the constructor (that is, <init>) of the class Connection is called. The tracing records when it started and when it finished. For operations that include the java.net package, the ORBRas logger prints also the number of the local port that was involved.

Comm traces:

An example of comm (wire) tracing.

Here is an example of comm tracing:

```
 // Summary of the message containing name-value pairs for the principal fields
OUT GOING:
Request Message // It is an out going request, therefore we are dealing with a client
Date:           31 January 2003 16:17:34 GMT
Thread Info:    P=852270:O=0:CT
Local Port:     4899 (0x1323)
Local IP:       9.20.178.136
Remote Port:    4893 (0x131D)
Remote IP:      9.20.178.136
GIOP Version:   1.2
Byte order:     big endian

Fragment to follow: No     // This is the last fragment of the request
Message size:   276 (0x114)
--

Request ID:         5  // Request Ids are in ascending sequence
Response Flag:      WITH_TARGET // it means we are expecting a reply to this request
Target Address:     0
Object Key:         length = 26 (0x1A)  // the object key is created by the server when exporting
                              // the servant and retrieved in the IOR using a naming service
                    4C4D4249 00000010 14F94CA4 00100000
                    00080000 00000000 0000
Operation:          message // That is the name of the method that the client invokes on the servant
Service Context:    length = 3 (0x3)   // There are three service contexts
Context ID:         1229081874 (0x49424D12) // Partner version service context. IBM only
Context data:       length = 8 (0x8)
                    00000000 14000005


Context ID:         1 (0x1)  // Codeset CORBA service context
Context data:       length = 12 (0xC)
                    00000000 00010001 00010100

Context ID:         6 (0x6)  // Codebase CORBA service context
Context data:       length = 168 (0xA8)
                    00000000 00000028 49444C3A 6F6D672E
                    6F72672F 53656E64 696E6743 6F6E7465
                    78742F43 6F646542 6173653A 312E3000
                    00000001 00000000 0000006C 00010200
                    0000000D 392E3230 2E313738 2E313336
                    00001324 0000001A 4C4D4249 00000010
                    15074A96 00100000 00080000 00000000
                    00000000 00000002 00000001 00000018
                    00000000 00010001 00000001 00010020
                    00010100 00000000 49424D0A 00000008
                    00000000 14000005
Data Offset:        11c
// raw data that goes in the wire in numbered rows of 16 bytes and the corresponding ASCII
decoding
0000: 47494F50 01020000 00000114 00000005    GIOP............
0010: 03000000 00000000 0000001A 4C4D4249    ............LMBI
0020: 00000010 14F94CA4 00100000 00080000    ......L.........
0030: 00000000 00000000 00000008 6D657373    ............mess
0040: 61676500 00000003 49424D12 00000008    age.....IBM.....
0050: 00000000 14000005 00000001 0000000C    ................
0060: 00000000 00010001 00010100 00000006    ................
0070: 000000A8 00000000 00000028 49444C3A    ...........(IDL:
0080: 6F6D672E 6F72672F 53656E64 696E6743    omg.org/SendingC
0090: 6F6E7465 78742F43 6F646542 6173653A    ontext/CodeBase:
00A0: 312E3000 00000001 00000000 0000006C    1.0............l
00B0: 00010200 0000000D 392E3230 2E313738    ........9.20.178
00C0: 2E313336 00001324 0000001A 4C4D4249    .136...$....LMBI
00D0: 00000010 15074A96 00100000 00080000    ......J.........
00E0: 00000000 00000000 00000002 00000001    ................
```

```
00F0: 00000018 00000000 00010001 00000001    ................
0100: 00010020 00010100 00000000 49424D0A    ... ........IBM.
0110: 00000008 00000000 14000005 00000000    ................
```

Note: The italic comments that start with a double slash have been added for clarity; they are not part of the traces.

In this example trace, you can see a summary of the principal fields that are contained in the message, followed by the message itself as it goes in the wire. In the summary are several field name-value pairs. Each number is in hexadecimal notation.

For details of the structure of a GIOP message, see the CORBA specification, chapters 13 and 15: http://www.omg.org/cgi-bin/doc?formal/99-10-07.

Client or server:

From the first line of the summary of the message, you can identify whether the host to which this trace belongs is acting as a server or as a client. OUT GOING means that the message has been generated on the workstation where the trace was taken and is sent to the wire.

In a distributed-object application, a server is defined as the provider of the implementation of the remote object to which the client connects. In this work, however, the convention is that a client sends a request while the server sends back a reply. In this way, the same ORB can be client and server in different moments of the rmi-iiop session.

The trace shows that the message is an outgoing request. Therefore, this trace is a client trace, or at least part of the trace where the application acts as a client.

Time information and host names are reported in the header of the message.

The Request ID and the Operation (“message” in this case) fields can be very helpful when multiple threads and clients destroy the logical sequence of the traces.

The GIOP version field can be checked if different ORBs are deployed. If two different ORBs support different versions of GIOP, the ORB that is using the more recent version of GIOP should fall back to a common level. By checking that field, however, you can easily check whether the two ORBs speak the same language.

Service contexts:

The header also records three service contexts, each consisting of a context ID and context data.

A service context is extra information that is attached to the message for purposes that can be vendor-specific such as the IBM Partner version 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 287.

In general, stop the application, enable the orb traces and restart the application. When the hang is reproduced, the partial traces that can be retrieved can be used by the IBM ORB service team to help understand where the problem is.

Running the client without the server running before the client is started:

An example of the error messages that are generated from this process.

This operation outputs:

```
(org.omg.CORBA.COMM_FAILURE)
Hello Client exception:
        org.omg.CORBA.COMM_FAILURE:minor code:1 completed:No
             at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:145)
             at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:77)
             at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
             at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:75)
             at com.ibm.rmi.corba.ClientDelegate.createRequest(ClientDelegate.java:440)
             at com.ibm.rmi.corba.ClientDelegate.is_a(ClientDelegate.java:571)
             at org.omg.CORBA.portable.ObjectImpl._is_a(ObjectImpl.java:74)
             at org.omg.CosNaming.NamingContextHelper.narrow(NamingContextHelper.java:58)
             com.sun.jndi.cosnaming.CNCtx.callResolve(CNCtx.java:327)
```

Client and server are running, but not naming service:

An example of the error messages that are generated from this process.

The output is:

```
Hello Client exception:Cannot connect to ORB
Javax.naming.CommunicationException:
        Cannot connect to ORB.Root exception is org.omg.CORBA.COMM_FAILURE  minor code:1 completed:No
               at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:145)
               at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:77)
               at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
               at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:75)
               at com.ibm.rmi.corba.ClientDelegate.createRequest(ClientDelegate.java:440)
               at com.ibm.rmi.corba.InitialNamingClient.resolve(InitialNamingClient.java:197)
               at com.ibm.rmi.corba.InitialNamingClient.cachedInitialReferences(InitialNamingClient.j
               at com.ibm.rmi.corba.InitialNamingClient.resolve_initial_references(InitialNamingClien
               at com.ibm.rmi.corba.ORB.resolve_initial_references(ORB.java:1269)
               .........
```

You must start the Java IDL name server before an application or applet starts that uses its naming service. Installation of the Java IDL product creates a script (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:

- 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, ensure the following:

- The scenario can be reproduced in a similar configuration.
- The JIT is disabled (see "JIT and AOT problem determination" on page 364).

Also:

- Disable additional CPUs.
- Disable Simultaneous Multithreading (SMT) where possible.
- Eliminate memory dependencies with the client or server. The lack of physical memory can be the cause of slow performance, apparent hangs, or crashes. To remove these problems, ensure that you have a reasonable headroom of memory.
- Check physical network problems (firewalls, comm links, routers, DNS name servers, and so on). These are the major causes of CORBA COMM_FAILURE exceptions. As a test, ping your own workstation name.
- If the application is using a database such as DB2, switch to the most reliable driver. For example, to isolate DB2 AppDriver, switch to Net Driver, which is slower and uses sockets, but is more reliable.

NLS problem determination

The JVM contains built-in support for different locales. This section provides an overview of locales, with the main focus on fonts and font management.

The topics are:

- "Overview of fonts"
- "Font utilities" on page 260
- "Common NLS problem and possible causes" on page 261

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

xfd (AIX)
Use the command `xfd -fn <physical font name>` in AIX to find out about the glyphs and their rendering capacity. For example: `Xfd -fn monotype-sansmonowt-medium-r-normal--*-%d-75-75-m-*-ibm-udcjp` brings up a window with all the glyphs that are in that font.

xlsfonts
Use **xlsfonts** to check whether a particular font is installed on the system. For example: `xlsfonts | grep ksc` will list all the Korean fonts in the system.

iconv
Use to convert the character encoding from one encoding to other. Converted text is written to standard output. For example: `iconv -f oldset -t newset [file ...]`

Options are:

-f oldset
Specifies the source codeset (encoding).

-t newset
Specifies the destination codeset (encoding).

file
The file that contain the characters to be converted; if no file is specified, standard input is used.

Common NLS problem and possible causes

A common NLS problem with potential solutions.

Why do I see a square box or ??? (question marks) in the SDK components?
This effect is caused mainly because Java is not able to find the correct font file to display the character. If a Korean character should be displayed, the system should be using the Korean locale, so that Java can take the correct font file. If you are seeing boxes or queries, check the following:

For Swing components:

1. Check your locale with `locale`
2. To change the locale, export `LANG=zh_TW` (for example)
3. If you know which font you have used in your application, such as serif, try to get the corresponding physical font by looking in the fontpath. If the font file is missing, try adding it there.

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

Attach API problem determination

This section helps you solve problems involving the Attach API.

The IBM Java Attach API uses shared semaphores, sockets, and file system artifacts to implement the attach protocol. Problems with these artifacts might adversely affect the operation of applications when they use the attach API.

Note: Error messages from agents on the target VM go to `stderr` or `stdout` for the target VM. They are not reported in the messages output by the attaching VM.

Deleting files in `/tmp`

The attach API depends on the contents of a common directory. By default the common directory is `/tmp/.com_ibm_tools_attach`. Problems are caused if you modify the common directory in one of the following ways:

- Deleting the common directory.
- Deleting the contents of the common directory.
- Changing the permissions of the common directory or any of its content.

If you do modify the common directory, possible effects include:

- Semaphore "leaks" might occur, where excessive numbers of unused shared semaphores are opened. You can remove the semaphores using the command:

  ```
  ipcrm -s <semid>
  ```

  Use the command to delete semaphores that have keys starting with "0xa1".
- The Java VMs might not be able to list existing target VMs.
- The Java VMs might not be able to attach to existing target VMs.
- The Java VM might not be able to enable its attach API.

If the common directory cannot be used, a Java VM attempts to recreate the common directory. However, the JVM cannot recreate the files related to currently executing VMs.

The VirtualMachine.attach(String id) method reports `AttachNotSupportedException: No provider for virtual machine id`

There are several possible reasons for this message:

- The target VM might be owned by another userid. The attach API can only connect a VM to a target VM with the same userid.
- The attach API for the target VM might not have launched yet. There is a short delay from when the Java VM launches to when the attach API is functional.
- The attach API for the target VM might have failed. Verify that the directory `/tmp/.com_ibm_tools_attach/<id>` exists, and that the directory is readable and writable by the userid.
- The target directory `/tmp/.com_ibm_tools_attach/<id>` might have been deleted.
- The attach API might not have been able to open the shared semaphore. To verify that there is at least one shared semaphore, use the command:

  ```
  ipcs -s
  ```

  If there is a shared semaphore, at least one key starting with "0xa1" appears in the output from the **`ipcs`** command.

  Note: The number of available semaphores is limited on systems which use System V IPC, including Linux, z/OS, and AIX.

The VirtualMachine.attach() method reports `AttachNotSupportedException`

There are several possible reasons for this message:

- The target process is dead or suspended.

- The target process, or the hosting system is heavily loaded. The result is a delay in responding to the attach request.
- The network protocol has imposed a wait time on the port used to attach to the target. The wait time might occur after heavy use of the attach API, or other protocols which use sockets. To check if any ports are in the `TIME_WAIT` state, use the command:

  ```
  netstat -a
  ```

The VirtualMachine.loadAgent(), VirtualMachine.loadAgentLibrary(),, or VirtualMachine.loadAgentPath() methods report `com.sun.tools.attach.AgentLoadException` or `com.sun.tools.attach.AgentInitializationException`

There are several possible reasons for this message:

- The JVMTI agent or the agent JAR file might be corrupted. Try loading the agent at startup time using the `-javaagent`, `-agentlib`, or `-agentpath` option, depending on which method reported the problem.
- The agent might be attempting an operation which is not available after VM startup.

A process running as root can see a target using AttachProvider.listVirtualMachines(), but attempting to attach results in an `AttachNotSupportedException`

A process can attach only to processes owned by the same user. To attach to a non-root process from a root process, first use the **su** command to change the effective UID of the attaching process to the UID of the target UID, before attempting to attach.

Using diagnostic tools

Diagnostics tools are available to help you solve your problems.

This section describes how to use the tools. The chapters are:

- "Overview of the available diagnostics" on page 264
- "Using dump agents" on page 271
- "Using Javadump" on page 286
- "Using Heapdump" on page 306
- "Using system dumps and the dump viewer" on page 315
- "Tracing Java applications and the JVM" on page 331
- "JIT and AOT problem determination" on page 364
- "Garbage Collector diagnostics" on page 375
- "Class-loader diagnostics" on page 383
- "Shared classes diagnostics" on page 385
- "Using the Reliability, Availability, and Serviceability Interface" on page 412
- "Using the HPROF Profiler" on page 425
- "Using the JVMTI" on page 429
- "Using the Diagnostic Tool Framework for Java" on page 442
- "Using JConsole" on page 449

Note: JVMPI is now a deprecated interface, replaced by JVMTI.

Overview of the available diagnostics

The diagnostics 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 271). 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 486 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). It can also be generated by calling 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 286 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 306 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 calling com.ibm.jvm.Dump.SystemDump() 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 diagnostics" on page 375 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 331.

Other data
Special options are available for producing diagnostic information relating to

- The JIT (see "JIT and AOT problem determination" on page 364)
- Class loading (see "Class-loader diagnostics" on page 383)
- Shared classes (see "Shared classes diagnostics" on page 385)

The SDK includes a JVMTI based profiling tool called HPROF, which produces information that can help you to determine the parts of an application that might be using system resources; see “Using the HPROF Profiler” on page 425 for more details.

The SDK also includes an unsupported, experimental tool called JConsole. This graphical monitoring tool is based on the java.lang.management API, which you can use to observe and, for some properties, control various aspects of the JVM's behavior.

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.

Heapdump analysis tooling:

A number of tools are available for working with Heapdumps.

See “Using Heapdump” on page 306 for more information.

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 315.

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 429. The HPROF tool provided with the SDK has been updated to use the JVMTI; see “Using the HPROF Profiler” on page 425.

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 429. 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 425.

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 442.

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 switched on 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 358 and "Tracing Java applications and the JVM" on page 331 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 412 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 Java runtime shut down. The Java runtime 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 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. 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 354.
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, 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 runtime 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 285.

System resource limits:

The JVM can fail to produce dumps because insufficient system resources are available.

The failure can be identified in the `JVMDUMP012E` message issued by the JVM, for example:

```
JVMDUMP032I JVM requested System dump using '/home/test/core.20090302.104740.
2171156.0001.dmp' in response to an event
JVMDUMP012E Error in System dump: cannot find core file: "No such file or
directory".
```

If you see this message, check **ulimit -Hc** is set high enough.

To check that your environment is configured to allow the JVM to produce dumps correctly, see "Setting up and checking your AIX environment" on page 213.

The JVM might still be unable to produce dumps under some circumstances. For example, if there is not enough memory available at the time the dump is taken, you might see the following message:

```
JVMDUMP006I Processing dump event "gpf", detail "" - please wait.
JVMDUMP032I JVM requested System dump using '/home/test/xa64/core.20090227.063319.
1343.0001.dmp' in response to an event
JVMDUMP012E Error in System dump: insufficient system resources to generate dump,
errno=12 "Cannot allocate memory"
```

In this case, follow the instructions in the "Disabling dump agents with **-Xrs** and **-Xrs:sync**" on page 285 section. The information helps you disable the JVM dump agents and configure the operating system to produce a system dump instead.

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"
- "Dump agents" on page 274
- "Dump events" on page 277
- "Advanced control of dump agents" on page 278
- "Dump agent tokens" on page 282
- "Default dump agents" on page 282
- "Removing dump agents" on page 283
- "Dump agent environment variables" on page 284
- "Signal mappings" on page 285
- "Platform specifics" on page 285

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.

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

-Xdump command-line option syntax

```
>>──-Xdump:──┬─help──────────────────────────────────────────┬────────────><
             ├─none──────────────────────────────────────────┤
             ├─events────────────────────────────────────────┤
             ├─request───────────────────────────────────────┤
             ├─tokens────────────────────────────────────────┤
             ├─dynamic───────────────────────────────────────┤
             ├─nofailover────────────────────────────────────┤
             ├─what──────────────────────────────────────────┤
             │ ┌─+───────┐                                   │
             │ ▼         │                                   │
             └───<agent>─┴─┬─────────────────────────────────┬┘
                           └─:─┬─help──────────────────────┬─┘
                               ├─none──────────────────────┤
                               ├─defaults──────────────────┤
                               │ ┌─,─────────────────────┐ │
                               │ │       ┌─+─────────┐   │ │
                               │ ▼       ▼           │   │ │
                               └───┬─events=───<event>─┤─┴─┘
                                   ├─exec=<command>────┤
                                   ├─file=<filename>───┤
                                   ├─filter=<filter>───┤
                                   ├─opts=<options>────┤
                                   ├─priority=<0-999>──┤
                                   ├─range=<ranges>────┤
                                   └─request=<requests>┘
```

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

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

For more information, see the manual for your shell.

Help options

These options provide usage and configuration information for dumps, as shown in the following table:

Command	Result
-Xdump:help	Display general dump help
-Xdump:events	List available trigger events
-Xdump:request	List additional VM requests
-Xdump:tokens	List recognized label tokens
-Xdump:what	Show registered agents on startup
-Xdump:<*agent*>:help	Provides detailed dump agent help
-Xdump:<*agent*>:defaults	Provides default settings for this agent

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 diagnostics.

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 275.
console	Basic thread dump to stderr.
system	Capture raw process image. See "Using system dumps and the dump viewer" on page 315.
tool	Run command-line program.
java	Write application summary. See "Using Javadump" on page 286.
heap	Capture heap graph. See "Using Heapdump" on page 306.
snap	Take a snap of the trace buffers.

Console dumps:

Console dumps are very basic dumps, in which the status of every Java thread is written to stderr.

In this example, the **range=1..1** suboption is used to control the amount of output to just one thread start (in this case, the start of the Signal Dispatcher thread).

```
 java -Xdump:console:events=thrstart+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".
```

See “Using system dumps and the dump viewer” on page 315 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)
```

Tool option:

The **tool** option allows external processes to be started when an event occurs.

The following example displays a simple message when the JVM stops. The `%pid` token is used to pass the pid of the process to the command. The list of available tokens can be printed with **-Xdump:tokens**, or found in “Dump agent tokens” on page 282. 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

 VMDUMP006I Processing dump event "vmstop", detail "#00000000" - please wait.
 JVMDUMP007I JVM Requesting Tool dump using 'echo process 254050 has finished'
 JVMDUMP011I Tool dump spawned process 344292
 process 254050 has finished
 JVMDUMP013I Processed dump event "vmstop", detail "#00000000".
```

By default, the **range** option is set to `1..1`. If you do not specify a range option for the dump agent the tool will be started once only. To start the tool every time the event occurs, set the **range** option to `1..0`. See "range option" on page 280 for more information.

Javadumps:

Javadumps are an internally generated and formatted analysis of the JVM, giving information that includes the Java threads present, the classes loaded, and heap statistics.

An example of producing a Javadump when a class is loaded:

```
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 286 for more information about analyzing a Javadump.

Heapdumps:

Heapdumps produce phd format files by default.

"Using Heapdump" on page 306 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 306 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 353 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 278 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**
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.

Event	Triggered when...	Filter operation
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.

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 275 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
```

You can use tokens to add context to dump file names. See "Dump agent tokens" on page 282 for more information.

The location for the dump is selected from these options, in this order:

1. The location specified on the command line.
2. The location specified by the relevant environment variable.
   - **IBM_JAVACOREDIR** for Javadump.
   - **IBM_HEAPDUMPDIR** for Heapdump.
   - **IBM_COREDIR** for system dump, .
   - **IBM_COREDIR** for snap traces, .
3. The current working directory of the JVM process.

If the directory does not exist, it 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.

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=vmstop,filter=*InvalidArgumentException#*.myVirtualMethod
```

In order to make this filter work, it must be changed to:

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#MyApplication.*
```

Class loading and exception events

You can filter class loading (load) and exception (throw, catch, uncaught, systhrow) events by Java class name:

```
-Xdump:java:events=throw,filter=java/lang/OutOfMem*
-Xdump:java:events=throw,filter=*MemoryError
-Xdump:java:events=throw,filter=*Memory*
```

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 square 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 square brackets.

vmstop event

You can filter the JVM shut down event by using one or more exit codes:

```
-Xdump:java:events=vmstop,filter=#129..192#-42#255
```

slow event

You can filter the slow event to change the time threshold from the default of 50 ms:

```
-Xdump:java:events=slow,filter=#300ms
```

You cannot set the filter to a time lower than the default time.

Other events

If you apply a filter to an event that does not support filtering, the filter is ignored.

opts option:

The Heapdump agent uses this option to specify the type of file to produce.

Heapdumps and the opts option

You can specify a PHD Heapdump, a classic text Heapdump, or both. For example:

–Xdump:heap:opts=PHD (default)
–Xdump:heap:opts=CLASSIC
–Xdump:heap:opts=PHD+CLASSIC

See “Enabling text formatted ("classic") Heapdumps” on page 307 for more information.

priority option:

One event can generate multiple dumps. The agents that produce each dump run sequentially and their order is determined by the **priority** keyword set for each agent.

Examination of the output from **-Xdump:what** shows that a gpf event produces a snap trace, a Javadump, and a system dump. In this example, the system dump 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 282

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 runtime. For example, you can direct Java dump files into a separate directory for each process, and guarantee unique files by adding a sequence number to the file name using:

```
-Xdump:java:defaults:file=dumps/%pid/javacore-%seq.txt
```

This option does not add a Javadump agent; it updates the default settings for Javadump agents. Further Javadump agents will then create dump files using this specification for filenames, unless overridden.

Note: Changing the defaults for a dump type will also affect the default agents for that dump type added by the JVM during initialization. For example if you change the default file name for Javadumps, that will change the file name used by the default Javadump agents. However, changing the default **range** option will not change the range used by the default Javadump agents, because those agents override the **range** option with specific values.

Dump agent tokens

Use tokens to add context to dump file names and to pass command-line arguments to the tool agent.

The tokens available are listed in the following table:

Token	Description
%Y	Year (4 digits)
%y	Year (2 digits)
%m	Month (2 digits)
%d	Day of the month (2 digits)
%H	Hour (2 digits)
%M	Minute (2 digits)
%S	Second (2 digits)
%pid	Process id
%uid	User name
%seq	Dump counter
%tick	msec counter
%home	Java home directory
%last	Last dump

Default dump agents

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

See "Removing dump agents" on page 283 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+abort+traceassert+corruptcache,
    label=/home/user/core.%Y%m%d.%H%M%S.%pid.%seq.dmp,
    range=1..0,
    priority=999,
    request=serial
----------------------
-Xdump:system:
    events=systhrow,
    filter=java/lang/OutOfMemoryError,
    label=/home/user/core.%Y%m%d.%H%M%S.%pid.%seq.dmp,
    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**.

Use this option so that you can subsequently specify a completely new dump configuration.

You can also remove dump agents of a particular type. For example, to turn off all Heapdumps (including default agents) but leave Javadump enabled, use the following option:

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

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

- If a user signal (`kill -QUIT`) is sent to the JVM, a brief listing of the Java threads including their stacks, status, and monitor information is written to stderr.
- If a crash occurs, information about the location of the crash, JVM options, and native and Java stack traces are written to stderr. A system dump is also written to the user's home directory.

Tip: Removing dump agents and specifying a new dump configuration can require a long set of command-line options. To reuse command-line options, save the new dump configuration in a file and use the **-Xoptionsfile** option. See "Specifying command-line options" on page 455 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
- *<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 488.

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 281.
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 you specify. Actions on other conditions are left 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	

Platform specifics

Dump output is written to different files, depending on the type of the dump. File names include a time stamp.

- **System dumps**: Output is written to a file named `core.%Y%m%d.%H%M%S.%pid.dmp`.
- **Javadumps**: Output is written to a file named `javacore.%Y%m%d.%H%M%S.%pid.%seq.txt`. See "Using Javadump" on page 286 for more information.
- **Heapdumps**: Output is written to a file named `heapdump.%Y%m%d.%H%M%S.%pid.phd`. See "Using Heapdump" on page 306 for more information.

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 so that any signals received are handled by the operating system.

Using the **-Xrs** command-line option prevents the Java runtime handling exception signals such SIGSEGV and SIGABRT. When the Java runtime signal handler is disabled, a SIGSEGV or GPF crash does not call the JVM dump agents. Instead, dumps are produced depending on the operating system.

For UNIX-based systems, you can use **-Xrs:sync** to disable signal handling for specific signals without disabling the **-Xdump** user event. The operating system handles a SIGSEGV signal, but the **-Xdump** agent can still be triggered by a SIGQUIT signal.

For more information about the **-Xrs** and **-Xrs:sync** options, see "JVM command-line options" on page 459.

Disabling dump agents in AIX

AIX produces a core file called `core` in the working directory when a process crashes. The file can be processed with **jextract** and analyzed with tools such as **jdmpview** and **DTFJ**. For the dump to be useful, configure your AIX environment to produce full core dumps. See "Setting up and checking your AIX environment" on page 213 for more details.

Using Javadump

Javadump produces files that contain diagnostic information related to the JVM and a Java application captured at a point during execution. For example, the information can be about the operating system, the application environment, threads, stacks, locks, and memory.

The exact contents depend on the platform on which you are running. By default, a Javadump occurs when the JVM terminates unexpectedly. A Javadump can also be triggered by sending specific signals to the JVM. Javadumps are human readable.

The preferred way to control the production of Javadumps is by enabling dump agents (see "Using dump agents" on page 271) using **–Xdump:java:** on application startup. You can also control Javadumps by the use of environment variables. See "Environment variables and Javadump" on page 306.

Default agents are in place that (if not overridden) create Javadumps when the JVM terminates unexpectedly or when an out-of-memory exception occurs. Javadumps are also triggered by default when specific signals are received by the JVM.

Note: Javadump is also known as **Javacore**. 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"
- "Triggering a Javadump" on page 287
- "Interpreting a Javadump" on page 288
- "Environment variables and Javadump" on page 306

Enabling a Javadump

Javadumps are enabled by default. You can turn off the production of Javadumps with **-Xdump:java:none**.

You are not recommended to turn off Javadumps because they are an essential diagnostics tool.

Use the **-Xdump:java** option to give more fine-grained control over the production of Javadumps. See "Using dump agents" on page 271 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:

A fatal native exception
: Not a Java Exception. A "fatal" exception is one that causes the JVM to stop. The JVM handles the event by producing a system dump followed by a snap trace file, a Javadump, and then terminating the process.

The JVM has insufficient memory to continue operation
: There are many reasons for running out of memory. See "Problem determination" on page 211 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 AIX is `SIGQUIT`. Use the command `kill -QUIT n` to send the signal to a process with process id (PID) `n`. Alternatively, press **CTRL+** in the shell window that started Java.

: The JVM 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" on page 286.

: 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 271 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 350

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 442 for more information.

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

```
1CIJAVAVERSION JRE 1.7.0 AIX ppc64-64 build 20110513_82212
(pap6470-20110517_01)
```

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. The components are highlighted for clarification.

```
NULL           ------------------------------------------------------------------
0SECTION       TITLE subcomponent dump routine
NULL           ================================
```

```
1TICHARSET     ISO8859-1
1TISIGINFO     Dump Event "vmstart" (00000001) received
1TIDATETIME    Date: 2011/05/17 at 12:04:43
1TIFILENAME    Javacore filename: /home/user1/javacore.20110517.120443.172218.0001.txt
1TIREQFLAGS    Request Flags: 0x81 (exclusive+preempt)
1TIPREPSTATE   Prep State: 0x106 (vm_access+exclusive_vm_access+)
NULL           ------------------------------------------------------------------
0SECTION       GPINFO subcomponent dump routine
NULL           ================================
2XHOSLEVEL     OS Level : AIX 7.1
2XHCPUS        Processors -
3XHCPUARCH       Architecture : ppc64
3XHNUMCPUS       How Many : 8
3XHNUMASUP       NUMA is either not supported or has been disabled by user
```

For the rest of the topics in this section, the tags are removed to aid readability.

TITLE, GPINFO, and ENVINFO sections:

At the start of a Javadump, the first three sections are the TITLE, GPINFO, and ENVINFO sections. They provide useful information about the cause of the dump.

The following example shows some output taken from a simple Java test program 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 ISO8859-1
| 1TISIGINFO Dump Event "gpf" (00002000) received
| 1TIDATETIME Date: 2011/01/17 at 16:44:20
| 1TIFILENAME Javacore filename: /home/test/javacore.20110117.164417.266582.0002.txt
| 1TIREQFLAGS Request Flags: 0x81 (exclusive+preempt)
| 1TIPREPSTATE Prep State: 0x0
| 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. If the failure was caused by a GPF, GPF information about the failure is provided, in this case showing that the protection exception was thrown from `MVSCR71D.dll`. The registers specific to the processor and architecture are also displayed.

```
| 0SECTION GPINFO subcomponent dump routine
| NULL ================================
| 2XHOSLEVEL OS Level : AIX 6.1
| 2XHCPUS Processors -
| 3XHCPUARCH Architecture : ppc
| 3XHNUMCPUS How Many : 8
| 3XHNUMASUP NUMA is either not supported or has been disabled by user
| NULL
| 1XHEXCPCODE J9Generic_Signal_Number: 00000004
| 1XHEXCPCODE Signal_Number: 0000000B
| 1XHEXCPCODE Error_Value: 00000000
| 1XHEXCPCODE Signal_Code: 00000033
| 1XHEXCPCODE Handler1: F0CF6D6C
| 1XHEXCPCODE Handler2: F0CF12C0
| NULL
| 1XHEXCPMODULE Module: ./libcrash.a
| 1XHEXCPMODULE Module_base_address: 31EE1000
```

```
|  NULL
|  1XHREGISTERS Registers:
|  2XHREGISTER R0: 00000000
|  2XHREGISTER R1: 3010E3A0
|  2XHREGISTER R2: 31EE2418
|  2XHREGISTER R3: 00000000
|  2XHREGISTER R4: D1A1C720
|  2XHREGISTER R5: 30C6D010
|  2XHREGISTER R6: 00000000
|  2XHREGISTER R7: 00000001
|  2XHREGISTER R8: 30C70AB0
|  ....
|  2XHREGISTER FPR0: 0000000000000000
|  2XHREGISTER FPR1: 3FF0000000000000
|  2XHREGISTER FPR2: 4330000080000001
|  2XHREGISTER FPR3: 0000000000000006
|  2XHREGISTER FPR4: 4330080000000000
|  2XHREGISTER FPR5: 40FF3B4800000000
|  2XHREGISTER FPR6: 40FF3B4800000000
|  2XHREGISTER FPR7: 433008000001F3B4
|  2XHREGISTER FPR8: 3FD5555560000000
|  ....
|  1XHFLAGS VM flags:00040000
```

The `GPINFO` section also refers to the vmState, recorded in the console output as `VM flags`. The vmState is the thread-specific state of what was happening in the JVM at the time of the crash. The value for vmState is a 32-bit hexadecimal number of the format MMMMSSSS, where MMMM is the major component and SSSS is component specific code.

Major component	Code number
INTERPRETER	0x10000
GC	0x20000
GROW_STACK	0x30000
JNI	0x40000
JIT_CODEGEN	0x50000
BCVERIFY	0x60000
RTVERIFY	0x70000
SHAREDCLASSES	0x80000

In the following example, the value for vmState is `VM flags:00040000`, which indicates a crash in the JNI component.

When the vmState major component is JNI, the crash might be caused by customer JNI code or by Java SDK JNI code. Check the Javadump to reveal which JNI routine was called at the point of failure. The JNI is the only component where a crash might be caused by customer code.

When the vmState major component is JIT_CODEGEN, see the information at "JIT and AOT problem determination" on page 364.

ENVINFO

Shows information about the JRE level that failed and details about the command line that launched the JVM process and the JVM environment in place.

```
|  0SECTION ENVINFO subcomponent dump routine
|  NULL ================================
|  1CIJAVAVERSION JRE 1.7.0 AIX ppc-32 build 20110111_72690 (pap3270-20110113_01)
|  1CIVMVERSION VM build R26_head_20110110_1913_B72666
```

```
1CIJITVERSION JIT enabled, AOT enabled - r11_20110110_18260
1CIGCVERSION GC - R26_head_20110110_1556_B72647
1CIRUNNINGAS Running as a standalone JVM
1CICMDLINE sdk/jre/bin/java -Djava.library.path=. Crash
1CIJAVAHOMEDIR Java Home Dir: /home/test/sdk/jre
1CIJAVADLLDIR Java DLL Dir: /home/test/sdk/jre/bin
1CISYSCP Sys Classpath: /home/test/sdk/jre/lib/ppc/default/jclSC170/vm.jar;....
1CIUSERARGS UserArgs:
2CIUSERARG -Xoptionsfile=/home/test/sdk/jre/lib/ppc/default/options.default
2CIUSERARG -Xjcl:jclse7b_26
2CIUSERARG -Dcom.ibm.oti.vm.bootstrap.library.path=/home/test/sdk/jre/lib/ppc/default....
2CIUSERARG -Dsun.boot.library.path=/home/test/sdk/jre/lib/ppc/default....
2CIUSERARG -Djava.library.path=/home/test/sdk/jre/lib/ppc/default....
2CIUSERARG -Djava.home=/home/test/sdk/jre
....
```

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 unlimited unlimited
2CIUSERLIMIT RLIMIT_CPU unlimited unlimited
2CIUSERLIMIT RLIMIT_DATA 2147483645 unlimited
2CIUSERLIMIT RLIMIT_FSIZE unlimited unlimited
2CIUSERLIMIT RLIMIT_NOFILE 2000 unlimited


1CIENVVARS Environment Variables
NULL ------------------------------------------------------------------------
2CIENVVAR _=sdk/jre/bin/java
2CIENVVAR LANG=en_GB
2CIENVVAR VISUAL=/bin/vi
2CIENVVAR SSH_TTY=/dev/pts/0
2CIENVVAR PATH=/usr/vacpp/bin:/usr/bin:/etc:/usr/sbin:/usr/ucb:.
2CIENVVAR EDITOR=/bin/vi
2CIENVVAR HOSTNAME=jalfrezi
2CIENVVAR LOCPATH=/usr/lib/nls/loc
2CIENVVAR FSEDITOR=/bin/vi
2CIENVVAR HISTSIZE=1000
2CIENVVAR HOME=/home/test
....
```

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 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.

```
0SECTION       NATIVEMEMINFO subcomponent dump routine
NULL           ================================
0MEMUSER
1MEMUSER       JRE: 1,646,859,372 bytes / 1141 allocations
1MEMUSER       |
2MEMUSER       +--VM: 1,644,024,980 bytes / 737 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Classes: 2,564,000 bytes / 99 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Memory Manager (GC): 1,639,582,776 bytes / 118 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Java Heap: 1,607,401,472 bytes / 1 allocation
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Other: 32,181,304 bytes / 117 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Threads: 1,217,340 bytes / 43 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Java Stack: 36,080 bytes / 5 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Native Stack: 1,114,112 bytes / 6 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Other: 67,148 bytes / 32 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Trace: 85,152 bytes / 214 allocations
2MEMUSER       |  |
3MEMUSER       |  +--JVMTI: 17,328 bytes / 13 allocations
2MEMUSER       |  |
3MEMUSER       |  +--JNI: 18,552 bytes / 44 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Port Library: 6,696 bytes / 58 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Other: 533,136 bytes / 148 allocations
1MEMUSER       |
2MEMUSER       +--JIT: 1,825,112 bytes / 128 allocations
2MEMUSER       |  |
3MEMUSER       |  +--JIT Code Cache: 524,320 bytes / 1 allocation
2MEMUSER       |  |
3MEMUSER       |  +--JIT Data Cache: 524,336 bytes / 1 allocation
2MEMUSER       |  |
3MEMUSER       |  +--Other: 776,456 bytes / 126 allocations
1MEMUSER       |
2MEMUSER       +--Class Libraries: 1,009,280 bytes / 276 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Harmony Class Libraries: 1,024 bytes / 1 allocation
2MEMUSER       |  |
3MEMUSER       |  +--VM Class Libraries: 1,008,256 bytes / 275 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--sun.misc.Unsafe: 69,688 bytes / 1 allocation
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Other: 938,568 bytes / 274 allocations
NULL
```

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.

This section also contains garbage collection history data, described in "Default memory management tracing" on page 333. 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 for tasks that use large amounts of memory. Example tasks are maintaining JIT caches, and storing Java classes. The Java runtime 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. 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:

  alloc The address of the top of the section of associated native memory that is currently in use. For some segment types, this address is the same as the end address.

  bytes The size of the attached native memory.

  end The end address of the attached storage.

  segment
  The address of the segment data structure.

  start The start address of the associated native memory.

  type The internal bit-field describing the characteristics of the associated native memory.

```
|  0SECTION       MEMINFO subcomponent dump routine
|  NULL           ================================
|  NULL
|  1STHEAPTYPE    Object Memory
|  NULL           id         start      end        size       space/region
|  1STHEAPSPACE   0x00497030     --         --         --     Generational
|  1STHEAPREGION  0x004A24F0 0x02850000 0x05850000 0x03000000 Generational/Tenured Region
|  1STHEAPREGION  0x004A2468 0x05850000 0x06050000 0x00800000 Generational/Nursery Region
|  1STHEAPREGION  0x004A23E0 0x06050000 0x06850000 0x00800000 Generational/Nursery Region
|  NULL
|  1STHEAPTOTAL   Total memory:           67108864 (0x04000000)
|  1STHEAPINUSE   Total memory in use:    33973024 (0x02066320)
|  1STHEAPFREE    Total memory free:      33135840 (0x01F99CE0)
|  NULL
|  1STSEGTYPE     Internal Memory
|  NULL           segment    start      alloc      end        type       size
|  1STSEGMENT     0x073DFC9C 0x0761B090 0x0761B090 0x0762B090 0x01000040 0x00010000
|         (lines removed for clarity)
|  1STSEGMENT     0x00497238 0x004FA220 0x004FA220 0x0050A220 0x00800040 0x00010000
|  NULL
|  1STSEGTOTAL    Total memory:           873412 (0x000D53C4)
|  1STSEGINUSE    Total memory in use:         0 (0x00000000)
|  1STSEGFREE     Total memory free:      873412 (0x000D53C4)
|  NULL
|  1STSEGTYPE     Class Memory
|  NULL           segment    start      alloc      end        type       size
|  1STSEGMENT     0x0731C858 0x0745C098 0x07464098 0x07464098 0x00010040 0x00008000
|          (lines removed for clarity)
|  1STSEGMENT     0x00498470 0x070079C8 0x07026DC0 0x070279C8 0x00020040 0x00020000
|  NULL
|  1STSEGTOTAL    Total memory:           2067100 (0x001F8A9C)
|  1STSEGINUSE    Total memory in use:    1839596 (0x001C11EC)
|  1STSEGFREE     Total memory free:       227504 (0x000378B0)
|  NULL
|  1STSEGTYPE     JIT Code Cache
|  NULL           segment    start      alloc      end        type       size
|  1STSEGMENT     0x004F9168 0x06960000 0x069E0000 0x069E0000 0x00000068 0x00080000
|  NULL
|  1STSEGTOTAL    Total memory:           524288 (0x00080000)
|  1STSEGINUSE    Total memory in use:    524288 (0x00080000)
|  1STSEGFREE     Total memory free:           0 (0x00000000)
|  NULL
|  1STSEGTYPE     JIT Data Cache
|  NULL           segment    start      alloc      end        type       size
|  1STSEGMENT     0x004F92E0 0x06A60038 0x06A6839C 0x06AE0038 0x00000048 0x00080000
|  NULL
|  1STSEGTOTAL    Total memory:           524288 (0x00080000)
|  1STSEGINUSE    Total memory in use:     33636 (0x00008364)
|  1STSEGFREE     Total memory free:      490652 (0x00077C9C)
|  NULL
|  1STGCHTYPE     GC History
|  3STHSTTYPE     15:18:14:901108829 GMT j9mm.134 -   Allocation failure end: newspace=7356368/8388608
|  oldspace=32038168/50331648 loa=3523072/3523072
|  3STHSTTYPE     15:18:14:901104380 GMT j9mm.470 -   Allocation failure cycle end: newspace=7356416/8388608
|  oldspace=32038168/50331648 loa=3523072/3523072
|  3STHSTTYPE     15:18:14:901097193 GMT j9mm.65 -   LocalGC end: rememberedsetoverflow=0
|  causedrememberedsetoverflow=0 scancacheoverflow=0 failedflipcount=0 failedflipbytes=0 failedtenurecount=0
|  failedtenurebytes=0 flipcount=11454 flipbytes=991056 newspace=7356416/8388608 oldspace=32038168/50331648
|  loa=3523072/3523072 tenureage=1
|  3STHSTTYPE     15:18:14:901081108 GMT j9mm.140 -   Tilt ratio: 50
|  3STHSTTYPE     15:18:14:893358658 GMT j9mm.64 -   LocalGC start: globalcount=3 scavengecount=24 weakrefs=0
|  soft=0 phantom=0 finalizers=0
|  3STHSTTYPE     15:18:14:893354551 GMT j9mm.63 -   Set scavenger backout flag=false
|  3STHSTTYPE     15:18:14:893348733 GMT j9mm.135 -   Exclusive access: exclusiveaccessms=0.002
|  meanexclusiveaccessms=0.002 threads=0 lastthreadtid=0x00495F00 beatenbyotherthread=0
|  3STHSTTYPE     15:18:14:893348391 GMT j9mm.469 -   Allocation failure cycle start: newspace=0/8388608
|  oldspace=38199368/50331648 loa=3523072/3523072 requestedbytes=48
```

```
|  3STHSTTYPE     15:18:14:893347364 GMT j9mm.133 -   Allocation failure start: newspace=0/8388608
|  oldspace=38199368/50331648 loa=3523072/3523072 requestedbytes=48
|  3STHSTTYPE     15:18:14:866523613 GMT j9mm.134 -   Allocation failure end: newspace=2359064/8388608
|  oldspace=38199368/50331648 loa=3523072/3523072
|  3STHSTTYPE     15:18:14:866519507 GMT j9mm.470 -   Allocation failure cycle end: newspace=2359296/8388608
|  oldspace=38199368/50331648 loa=3523072/3523072
|  3STHSTTYPE     15:18:14:866513004 GMT j9mm.65 -   LocalGC end: rememberedsetoverflow=0
|  causedrememberedsetoverflow=0 scancacheoverflow=0 failedflipcount=5056 failedflipbytes=445632
|  failedtenurecount=0 failedtenurebytes=0 flipcount=9212 flipbytes=6017148 newspace=2359296/8388608
|  oldspace=38199368/50331648 loa=3523072/3523072 tenureage=1
|  3STHSTTYPE     15:18:14:866493839 GMT j9mm.140 -   Tilt ratio: 64
|  3STHSTTYPE     15:18:14:859814852 GMT j9mm.64 -   LocalGC start: globalcount=3 scavengecount=23 weakrefs=0
|  soft=0 phantom=0 finalizers=0
|  3STHSTTYPE     15:18:14:859808692 GMT j9mm.63 -   Set scavenger backout flag=false
|  3STHSTTYPE     15:18:14:859801848 GMT j9mm.135 -   Exclusive access: exclusiveaccessms=0.004
|  meanexclusiveaccessms=0.004 threads=0 lastthreadtid=0x00495F00 beatenbyotherthread=0
|  3STHSTTYPE     15:18:14:859801163 GMT j9mm.469 -   Allocation failure cycle start: newspace=0/10747904
|  oldspace=38985800/50331648 loa=3523072/3523072 requestedbytes=232
|  3STHSTTYPE     15:18:14:859800479 GMT j9mm.133 -   Allocation failure start: newspace=0/10747904
|  oldspace=38985800/50331648 loa=3523072/3523072 requestedbytes=232
|  3STHSTTYPE     15:18:14:652219028 GMT j9mm.134 -   Allocation failure end: newspace=2868224/10747904
|  oldspace=38985800/50331648 loa=3523072/3523072
|  3STHSTTYPE     15:18:14:650796714 GMT j9mm.470 -   Allocation failure cycle end: newspace=2868224/10747904
|  oldspace=38985800/50331648 loa=3523072/3523072
|  3STHSTTYPE     15:18:14:650792607 GMT j9mm.475 -   GlobalGC end: workstackoverflow=0 overflowcount=0
|  memory=41854024/61079552
|  3STHSTTYPE     15:18:14:650784052 GMT j9mm.90 -   GlobalGC collect complete
|  3STHSTTYPE     15:18:14:650780971 GMT j9mm.57 -   Sweep end
|  3STHSTTYPE     15:18:14:650611567 GMT j9mm.56 -   Sweep start
|  3STHSTTYPE     15:18:14:650610540 GMT j9mm.55 -   Mark end
|  3STHSTTYPE     15:18:14:645222792 GMT j9mm.54 -   Mark start
|  3STHSTTYPE     15:18:14:645216632 GMT j9mm.474 -   GlobalGC start: globalcount=2
|              (lines removed for clarity)
|  NULL
|  NULL           ------------------------------------------------------------------------
```

Locks, monitors, and deadlocks (LOCKS):

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

A lock, also referred to as a monitor, prevents more than one entity from accessing a shared resource. Each object in Java has an associated lock, obtained by using a synchronized block or method. In the case of the JVM, threads compete for various resources in the JVM and locks on Java objects.

This example was taken from a deadlock test program where two threads “DeadLockThread 0” and “DeadLockThread 1” were unsuccessfully attempting to synchronize (Java keyword) on two java/lang/Integers.

You can see in the example (highlighted) that “DeadLockThread 1” has locked the object instance java/lang/Integer@004B2290. The monitor has been created as a result of a Java code fragment looking like “synchronize(count0)”, and this monitor has “DeadLockThread 1” waiting to get a lock on this same object instance (count0 from the code fragment). Below the highlighted section is another monitor locked by “DeadLockThread 0” that has “DeadLockThread 1” waiting.

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

```
------------------------------------------------------------------------
LOCKS subcomponent dump routine
===============================
```

```
    Monitor pool info:
     Current total number of monitors: 2

Monitor Pool Dump (flat & inflated object-monitors):
      sys_mon_t:0x00039B40 infl_mon_t: 0x00039B80:
       java/lang/Integer@004B22A0/004B22AC: Flat locked by "DeadLockThread 1"
                                                (0x41DAB100), entry count 1
            Waiting to enter:
                "DeadLockThread 0" (0x41DAAD00)      sys_mon_t:0x00039B98 infl_mon_t: 0x00039BD8:
       java/lang/Integer@004B2290/004B229C: Flat locked by "DeadLockThread 0"
                                                          (0x41DAAD00), entry count 1
            Waiting to enter:
                "DeadLockThread 1" (0x41DAB100)
JVM System Monitor Dump (registered monitors):
          Thread global lock (0x00034878): <unowned>
          NLS hash table lock (0x00034928): <unowned>
          portLibrary_j9sig_async_monitor lock (0x00034980): <unowned>
          Hook Interface lock (0x000349D8): <unowned>
            < lines removed for brevity >

=======================
Deadlock detected !!!
---------------------

  Thread "DeadLockThread 1" (0x41DAB100)
    is waiting for:
      sys_mon_t:0x00039B98 infl_mon_t: 0x00039BD8:
      java/lang/Integer@004B2290/004B229C:
    which is owned by:
  Thread "DeadLockThread 0" (0x41DAAD00)
    which is waiting for:
      sys_mon_t:0x00039B40 infl_mon_t: 0x00039B80:
      java/lang/Integer@004B22A0/004B22AC:
    which is owned by:
  Thread "DeadLockThread 1" (0x41DAB100)
```

Threads and stack trace (THREADS):

For the application programmer, one of the most useful pieces of a Java dump is the THREADS section. This section shows a list of Java threads, 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:

```
"main" J9VMThread:0x41D11D00, j9thread_t:0x003C65D8, java/lang/Thread:0x40BD6070, state:CW, prio=5
 (native thread ID:0xA98, native priority:0x5, native policy:UNKNOWN)
 Java callstack:
 at java/lang/Thread.sleep(Native Method)
 at java/lang/Thread.sleep(Thread.java:862)
 at mySleep.main(mySleep.java:31)
```

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. The properties on the second 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 301.

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:0x41481900, j9thread_t:0x002A54A4, java/lang/Thread:0x004316B8,
state:CW, prio=5
3XMTHREADINFO1            (native thread ID:0x904, native priority:0x5, native policy:UNKNOWN)
3XMTHREADINFO3           Java callstack:
4XESTACKTRACE                at java/lang/String.getChars(String.java:667)
4XESTACKTRACE                at java/lang/StringBuilder.append(StringBuilder.java:207)
```

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 281 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:

```
 "Attach handler" TID:0x02288600, j9thread_t:0x00295EB8, state:R, prio=5
            (native thread ID:0x3FC4, native priority:0x5, native policy:UNKNOWN)
           Java callstack:
               at com/ibm/tools/attach/javaSE/IPC.waitSemaphoreImpl(Native Method)
               at com/ibm/tools/attach/javaSE/IPC.waitSemaphore(IPC.java:191)
               at com/ibm/tools/attach/javaSE/AttachHandler.waitForNotification(AttachHandler.java:174)
               at com/ibm/tools/attach/javaSE/AttachHandler.run(AttachHandler.java:149)
           Native callstack:
               KiFastSystemCallRet+0x0 (0x7C82860C [ntdll+0x2860c])
               WaitForSingleObject+0x12 (0x77E61C8D [kernel32+0x21c8d])
               j9shsem_wait+0x100 (j9shsem.c:228, 0x7FE0F450 [J9PRT26+0xf450])
               Java_com_ibm_tools_attach_javaSE_IPC_waitSemaphoreImpl+0x66 (attach.c:356, 0x7FAE4A46
[jclscar_26+0x4a46])
               VMprJavaSendNative+0x4b0 (jnisend.asm:1147, 0x7FE52CA0 [j9vm26+0x22ca0])
               javaProtectedThreadProc+0xd7 (vmthread.c:1620, 0x7FE7F697 [j9vm26+0x4f697])
               j9sig_protect+0x42 (j9signal.c:144, 0x7FE117B2 [J9PRT26+0x117b2])
               javaThreadProc+0x5c (vmthread.c:261, 0x7FE7D14C [j9vm26+0x4d14c])
               thread_wrapper+0x133 (j9thread.c:975, 0x7FFA1FE3 [J9THR26+0x1fe3])
               _threadstart+0x6c (thread.c:196, 0x7C34940F [msvcr71+0x940f])
               GetModuleHandleA+0xdf (0x77E6482F [kernel32+0x2482f])
```

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. The last four frames of the native stack for JIT Compilation Thread show the behavior when the heap used for stack frames is exhausted. The raw instruction pointer address is used instead.

```
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])
4XENATIVESTACK               (0x00439F93)
4XENATIVESTACK               (0x7FFA1FE3)
4XENATIVESTACK               (0x7C34940F)
4XENATIVESTACK               (0x77E6482F)
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 diagnostics, 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 403 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 284 for more information.
IBM_JAVADUMP_OUTOFMEMORY	By setting this environment variable to false, you disable Javadumps for an out-of-memory exception.

Using Heapdump

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

This dump is stored in a Portable Heap Dump (PHD) file, a compressed binary format. You can use various tools on the Heapdump output to analyze the composition of the objects on the heap and (for example) help to find the objects that are controlling large amounts of memory on the Java heap and the reason why the Garbage Collector cannot collect them.

This chapter describes:

- "Getting Heapdumps"
- "Available tools for processing Heapdumps" on page 307
- "Using **-Xverbose:gc** to obtain heap information" on page 307
- "Environment variables and Heapdump" on page 307
- "Text (classic) Heapdump file format" on page 308

Getting Heapdumps

By default, a Heapdump is produced when the Java heap is exhausted. Heapdumps can be generated in other situations by use of **-Xdump:heap**.

See "Using dump agents" on page 271 for more detailed information about generating dumps based on specific events. Heapdumps can also be generated programmatically by use of the com.ibm.jvm.Dump.HeapDump() method from inside the application code.

To see which events will trigger a dump, use **-Xdump:what**. See “Using dump agents” on page 271 for more information.

By default, Heapdumps are produced in PHD format. To produce Heapdumps in text format, see “Enabling text formatted ("classic") Heapdumps.”

Environment variables can also affect the generation of Heapdumps (although this is a deprecated mechanism). See “Environment variables and Heapdump” 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 271). 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 284 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.

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: J2RE 7.0 IBM J9 2.6 Windows XP x86-32 build 20100303_054799
```

Object records

Object records are multiple records, one for each object instance on the heap, providing object address, size, type, and references from the object.

```
<object address, in hexadecimal> [<length in bytes of object instance, in decimal>]
OBJ <object type> <class block reference, in hexadecimal>
<heap reference, in hexadecimal <heap reference, in hexadecimal> ...
```

The object address and heap references are in the heap, but the class block address is outside the heap. All references found in the object instance are listed, including refereces that are null values. The object type is either a class name including package or a primitive array or class array type, shown by its standard JVM type signature, see "Java VM type signatures" on page 310. Object records can also contain additional class block references, typically in the case of reflection class instances.

Examples:

An object instance, length 28 bytes, of type java/lang/String:

```
0x00436E90 [28] OBJ java/lang/String
```

A class block address of java/lang/String, followed by a reference to a char array instance:

```
0x415319D8 0x00436EB0
```

An object instance, length 44 bytes, of type char array:

```
0x00436EB0 [44] OBJ [C
```

A class block address of char array:

```
0x41530F20
```

An object of type array of java/util/Hashtable Entry inner class:

```
0x004380C0 [108] OBJ [Ljava/util/Hashtable$Entry;
```

An object of type java/util/Hashtable Entry inner class:

```
0x4158CD80 0x00000000 0x00000000 0x00000000 0x00000000 0x00421660 0x004381C0
0x00438130 0x00438160 0x00421618 0x00421690 0x00000000 0x00000000 0x00000000
0x00438178 0x004381A8 0x004381F0 0x00000000 0x004381D8 0x00000000 0x00438190
0x00000000 0x004216A8 0x00000000 0x00438130 [24] OBJ java/util/Hashtable$Entry
```

A class block address and heap references, including null references:

```
0x4158CB88 0x004219B8 0x004341F0 0x00000000
```

Class records

Class records are multiple records, one for each loaded class, providing class block address, size, type, and references from the class.

```
<class block address, in hexadecimal> [<length in bytes of class block, in decimal>]
CLS <class type>
<class block reference, in hexadecimal> <class block reference, in hexadecimal> ...
<heap reference, in hexadecimal> <heap reference, in hexadecimal>...
```

The class block address and class block references are outside the heap, but the class record can also contain references into the heap, typically for static class data members. All references found in the class block are listed, including those that are null values. The class type is either a class name including package or a primitive array or class array type, shown by its standard JVM type signature, see “Java VM type signatures” on page 310.

Examples:

A class block, length 168 bytes, for class java/lang/Runnable:

```
0x41532E68 [168] CLS java/lang/Runnable
```

References to other class blocks and heap references, including null references:

```
0x4152F018 0x41532E68 0x00000000 0x00000000 0x00499790
```

A class block, length 168 bytes, for class java/lang/Math:

```
0x00000000 0x004206A8 0x00420720 0x00420740 0x00420760 0x00420780 0x004207B0
0x00421208 0x00421270 0x00421290 0x004212B0 0x004213C8 0x00421458 0x00421478
0x00000000 0x41589DE0 0x00000000 0x4158B340 0x00000000 0x00000000 0x00000000
0x4158ACE8 0x00000000 0x4152F018 0x00000000 0x00000000 0x00000000
```

Trailer record 1

Trailer record 1 is a single record containing record counts.

```
// Breakdown - Classes: <class record count, in decimal>,
Objects: <object record count, in decimal>,
ObjectArrays: <object array record count, in decimal>,
PrimitiveArrays: <primitive array record count, in decimal>
```

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.

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

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

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

PHD object records:

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

Short object record

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

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

Medium object record

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

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

- If all objects are hashed, a “short” containing the hashcode.
- The array of references. For more information, see the description in the short record format.

Long object record

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

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

PHD array records:

PHD array records can cover primitive arrays and object arrays.

Primitive array record

The primitive array record contains:

- The 1 byte tag. The third bit (0x20) is set and the following 5 bits in descending order contain:
  - 3 bits containing the array type. The array type values are:
    - 0 = bool
    - 1 = char
    - 2 = float
    - 3 = double
    - 4 = byte
    - 5 = short
    - 6 = int
    - 7 = long
  - 2 bits indicating the length of the array size and also the length of the gap. These values apply:
    - 0 indicates a “byte”.
    - 1 indicates a “short”.

- 2 indicates an “int”.
- 3 indicates a “long”.

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

Long primitive array record

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

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

Object array record

The object array record format is:

- The 1 byte tag containing the value 5.
- A “byte” containing flags with these bits in descending order:
  - 2 bits to indicate whether the gap is “byte”, “short”, “int” or “long”.
  - 2 bits indicating the size of each reference. For more information, see the description in the short record format.
  - 2 unused bits.
  - 1 bit indicating if the object was hashed and moved. If this bit is set, the record includes the hashcode.
  - 1 bit indicating if the object was hashed.
- A “byte”, “short”, “int” or “long” containing the gap between the address of this object and the address of the preceding object. For more information, see the description in the short record format.
- A “word” containing the address of the class of the objects in the array. Object array records do not update the class cache.

- If all objects are hashed, a “short” containing the hashcode. If the hashed and moved bit is set in the records flag, this field contains an “int”.
- An “int” containing the length of the array of references.
- The array of references. For more information, see the description in the short record format.

Object array record (revised) - from PHD version 5

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

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

This record type was added in PHD version 5.

PHD class records:

The PHD class record encodes a class object.

Class record

The format of a class record is:

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

Using system dumps and the dump viewer

The JVM can generate native system dumps, also known as core dumps, under configurable conditions. System dumps are typically quite large. Most tools used to analyze system dumps are also platform-specific; for example, **windbg** on Windows and **gdb** on Linux.

Dump agents are the primary method for controlling the generation of system dumps. See “Using dump agents” on page 271 for more information. To maintain

backwards compatibility, the JVM supports the use of environment variables for system dump triggering. See “Dump agent environment variables” on page 284 for more information.

The dump viewer is a cross-platform tool to analyze system dumps at the JVM level rather than at the platform level.

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 317

Overview of system dumps

The JVM can produce system dumps in response to specific events. A system dump is a raw binary dump of the process memory when the dump agent is triggered by a failure or by an event for which a dump is requested.

Generally, you use a tool to examine the contents of a system dump. A dump viewer tool is provided in the SDK, as described in this section, or you could use a platform-specific debugger, such as dbx, to examine the dump.

For dumps triggered by a General Protection Fault (GPF), dumps produced by the JVM contain some context information that you can read. You can find this failure context information by searching in the dump for the eye-catcher

```
J9Generic_Signal_Number
```

For example:

```
J9Generic_Signal_Number=00000004 ExceptionCode=c0000005 ExceptionAddress=7FAB506D ContextFlags=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 271 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.

Dump viewer: `jdmpview`

The dump viewer is a tool that allows you to examine the contents of system dumps produced from the JVM. The dump viewer requires metadata created by the **jextract** 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.

-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** guesses the names of the files in the compressed file.

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.

You can significantly improve the performance of **jdmpview** against large dumps by ensuring that your system has enough memory available to avoid paging. On large dumps (that is, ones with large numbers of objects on the heap), you might have to run **jdmpview** using the **-Xmx** option to increase the maximum heap available:

```
jdmpview  -J-Xmx<n> -zip <zip file>
```

To pass command-line arguments to the JVM, you must prefix them with **-J**.

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.

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 28 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.

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.

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 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 number of items to display and unit size ('b' for byte (8 bit), 'h' for half word (16 bit), 'w' for word (32 bit), 'g' for giant word (64 bit)) to sub-command (for example x/12bd). This is like the use of the **x/** command in **gdb** (including use of defaults).

x/J [*<0xaddr>***|***<class_name>***]**
Displays information about a particular object, or all objects of a class. If *<class_name>* is supplied, all static fields with their values are shown, followed by all objects of that class with their fields and values. If an object address (in hex) is supplied, static fields for that object's class are not shown; the other fields and values of that object are printed along with its address.

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

x/D *<0xaddr>*
Displays the integer at the specified address, adjusted for the hardware architecture this dump file is from. For example, the file might be from a big endian architecture.

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

x/X *<0xaddr>*
Displays the hex value of the bytes at the specified address, adjusted for the hardware architecture this dump file is from. For example, the file might be from a big endian architecture.

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

x/K *<0xaddr>*
Where the size is defined by the pointer size of the architecture, this parameter

shows the value of each section of memory. The output is adjusted for the hardware architecture this dump file is from, starting at the specified address. It also displays a module with a module section and an offset from the start of that module section in memory if the pointer points to that module section. If no symbol is found, it displays a "*" and an offset from the current address if the pointer points to an address in 4KB (4096 bytes) of the current address. Although this command can work on an arbitrary section of memory, it is probably more useful on a section of memory that refers to a stack frame. To find the memory section of a thread stack frame, use the **info thread** command.

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

Example session:

This example session illustrates a selection of the commands available and their use.

In the example session, some lines have been removed for clarity (and terseness). Some comments (contained inside braces) are included to explain various aspects with some comments on individual lines looking like:

```
{ comment }
```

User input is prefaced by a ">".

```
jammy@jtcid:~/tests$ jdmpview -core core.20070323.174921.22639.dmp
Loading image from DTFJ...

DTFJView version 1.0.14
Using DTFJ API version 1.1
For a list of commands, type "help"; for how to use "help", type "help help"
> help

        info
         thread   displays information about Java and native threads
         system   displays information about the system the core dump is from
         class    prints inheritance chain and other data for a given class
         proc     displays threads, command line arguments, environment variables, and shared modules of current process
         jitm     displays JIT'ed methods and their addresses
         ls       outputs a list of available monitors and locked objects
         sym      outputs a list of available modules
         mmap     outputs a list of all memory segments in the address space
         heap     displays information about Java heaps
        hexdump   outputs a section of memory in a hexdump-like format
        +         displays the next section of memory in hexdump-like format
        -         displays the previous section of memory in hexdump-like format
        find      searches memory for a given string
        deadlock  displays information about deadlocks if there are any
        set
         logging  configures several logging-related parameters, starts/stops logging
        show
         logging  displays the current values of logging settings
        quit      exits the core file viewing tool
        whatis    gives information about what is stored at the given memory address
        cd        changes the current working directory, used for log files
        pwd       displays the current working directory
        findnext  finds the next instance of the last string passed to "find"
        findptr   searches memory for the given pointer
        help      displays list of commands or help for a specific command
        x/        works like "x/" in gdb (including use of defaults): passes number of items to display and unit size ('b'
for byte, 'h' for halfword, 'w' for word, 'g' for giant word) to sub-command (ie. x/12bd)
         j        displays information about a particular object or all objects of a class
         d        displays the integer at the specified address
         x        displays the hex value of the bytes at the specified address
         k        displays the specified memory section as if it were a stack frame

> set logging file log.txt
```

```
        log file set to "log.txt"

> set logging on

        logging turned on; outputting to "/home/jammy/tests/log.txt"

> show logging

        set_logging = on
        set_logging_file = "log.txt"
        set_logging_overwrite = off
        set_logging_redirect = off

        current_logging_file = "/home/jammy/tests/log.txt"

>info thread *
{ Only one native thread is shown because under linux, the system dump is written after performing a fork }

        native threads for address space # 0
         process id: 22650

          thread id: 22650
           registers:
            cs       = 0x00000073   ds       = 0x0000007b   eax      = 0x00000000   ebp      = 0xb7d60994
            ebx      = 0xb7e353dc   ecx      = 0x00000002   edi      = 0xb7d60dbc   edx      = 0x00000001
            efl      = 0x00010296   eip      = 0xb7e1dc50   es       = 0xc010007b   esi      = 0xb7d60dcd
            esp      = 0xb7d6055c   fs       = 0x00000000   gs       = 0x00000033   ss       = 0x0000007b
           stack sections:
            0xb7562000 to 0xb7d62000 (length 0x800000)
           stack frames:
            ===Lines Removed===
           properties:

           associated Java thread: <no associated Java thread>

        Java threads not associated with known native threads:

            name:          main
            Thread object: 0x96fe3700
            JVMTI state:   ALIVE RUNNABLE
            Thread.State:  RUNNABLE
            Priority:      5
            monitor:       <no associated monitor>
            thread frames:
             bp: 0x081254e8    method: void com/ibm/jvm/Dump.SystemDumpImpl()    filename:line: <data unavailable>:
<data unavailable>
             bp: 0x081254f4    method: void com/ibm/jvm/Dump.SystemDump()    filename:line: <data unavailable>:
<data unavailable>
             bp: 0x0812551c    method: void DeadlockCreator.main(String[])    filename:line: DeadlockCreator.java:59

            name:          JIT Compilation Thread
            Thread object: 0x9702fe20
            JVMTI state:   ALIVE WAITING IN_OBJECT_WAIT
            Thread.State:  <no matching state>
            Priority:      10
            monitor:       waiting to be notified on "JIT-CompilationQueueMonitor" with ID <unavailable>
            thread frames: <no frames to print>

            name:          Signal Dispatcher
            Thread object: 0x97029d20
            JVMTI state:   ALIVE RUNNABLE
            Thread.State:  RUNNABLE
            Priority:      5
            monitor:       <no associated monitor>
            thread frames:
             bp: 0x08286ddc    method: int com/ibm/misc/SignalDispatcher.waitForSignal()    filename:line:
SignalDispatcher.java:<data unavailable>
             bp: 0x08286dfc    method: void com/ibm/misc/SignalDispatcher.run()    filename:line: SignalDispatcher.java:64

            name:          Gc Slave Thread
            Thread object: 0x970367c0
            JVMTI state:   ALIVE WAITING IN_OBJECT_WAIT
            Thread.State:  <no matching state>
            Priority:      5
            monitor:       waiting to be notified on "MM_ParallelDispatcher::slaveThread" with ID <unavailable>
            thread frames: <no frames to print>

            name:          Thread-1
            Thread object: 0x97064730
```

```
            JVMTI state:   ALIVE BLOCKED_ON_MONITOR_ENTER
            Thread.State:  BLOCKED
            Priority:      5
            monitor:       waiting to enter "(un-named monitor @0x9706b880 for object @0x9706b880)" with object 0x9706b880
            thread frames:
             bp: 0x083902cc    method: void DeadlockThreadA.syncMethod(DeadlockThreadA)    filename:line: DeadlockThreadA.
java:38
             bp: 0x083902dc    method: void DeadlockThreadA.run()    filename:line: DeadlockThreadA.java:51

            name:          Thread-2
            Thread object: 0x97065808
            JVMTI state:   ALIVE BLOCKED_ON_MONITOR_ENTER
            Thread.State:  BLOCKED
            Priority:      5
            monitor:       waiting to enter "(un-named monitor @0x9706b808 for object @0x9706b808)" with object 0x9706b808
            thread frames:
             bp: 0x083953f4    method: void DeadlockThreadB.syncMethod(DeadlockThreadB)    filename:line:
DeadlockThreadB.java:38
             bp: 0x08395404    method: void DeadlockThreadB.run()    filename:line: DeadlockThreadB.java:51
```

>info system

```
System:                Linux
System Memory:        2650529792 bytes
Virtual Machine(s):
        Runtime #1:
                Java(TM) 2 Runtime Environment, Standard Edition(build 2.4)
                IBM J9 VM(J2RE 1.6.0 IBM J9 2.4 Linux x86-32 jvmxi3260-20070321_12042 (JIT enabled)
                J9VM - 20070321_12042_lHdSMR
                JIT  - dev_20070321_1807_dev
                GC   - 20070319_AA)
```

> info class

```
        runtime #1 - version: Java(TM) 2 Runtime Environment, Standard Edition(build 2.4)
IBM J9 VM(J2RE 1.6.0 IBM J9 2.4 Linux x86-32 jvmxi3260-20070321_12042 (JIT enabled)
J9VM - 20070321_12042_lHdSMR
JIT  - dev_20070321_1807_dev
GC   - 20070319_AA)
               instances      total size  class name
                       1              24  java/util/BitSet
                      ===Lines Removed===
                      18             792  sun/misc/URLClassPath$JarLoader

         Total number of objects: 7765
         Total size of objects: 523441
```

> info class java/lang/String

```
        runtime #1 - version: Java(TM) 2 Runtime Environment, Standard Edition(build 2.4)
IBM J9 VM(J2RE 1.6.0 IBM J9 2.4 Linux x86-32 jvmxi3260-20070321_12042 (JIT enabled)
J9VM - 20070321_12042_lHdSMR
JIT  - dev_20070321_1807_dev
GC   - 20070319_AA)
name = java/lang/String

        ID = 0x80bd778    superID = 0x80ba778
        classLoader = 0x81f0370    modifers: public final synchronized

        number of instances:     2320
        total size of instances: 64960 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> @ 0x96fe32d0
            private static final char[] ascii = <object> @ 0x96fe32e0
            private static char[] startCombiningAbove = <object> @ 0x96fe3af0
            private static char[] endCombiningAbove = <object> @ 0x96fe3b80
            private static final char[] upperValues = <object> @ 0x96fe3c10
            private static final java.io.ObjectStreamField[] serialPersistentFields = <object> @ 0x96fe33f0
```

```
          non-static fields for "java/lang/String"
            private final char[] value
            private final int offset
            private final int count
            private int hashCode


Methods......

Bytecode range(s): 80fa634 -- 80fa64a:  public void <init>()
Bytecode range(s): 80fa660 -- 80fa69e:  private void <init>(String, char)
===Lines Removed===
Bytecode range(s): :  public native String intern()
===Lines Removed===
Bytecode range(s): 80fc82c -- 80fc835:  public volatile int compareTo(Object)
Bytecode range(s): 80fc84c -- 80fc896:  static void <clinit>()

> info proc

        address space # 0

         Thread information for current process:
          Thread id: 22650

         Command line arguments used for current process:
          ../sdk/ibm-java2-i386-60/jre/bin/java DeadlockCreator

         Environment variables for current process:
          JAVA_HOME=/usr/lib/ibm-java2-i386-50
          HOME=/home/jammy
          SHELL=/bin/bash
          PWD=/home/jammy/tests
          PS1=${debian_chroot:+($debian_chroot)}\u@\h:\w\$
          PATH=/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/share/lps/bin
          TERM=xterm
          ===Lines Removed===

> info jitm

        start=0x96647e7c  end=0x96647e90   java/lang/Object::<init>()V
        start=0x9664669c  end=0x96646eec   java/lang/J9VMInternals::verify(Ljava/lang/Class;)V
        ===Lines Removed===
        start=0x966499d4  end=0x96649aca   java/util/zip/Inflater::init(Z)J
        start=0x966491d4  end=0x966492db   java/util/zip/Inflater::inflateBytes([BII)I

> info ls
(un-named monitor @0x8374178 for object @0x8374178)
        owner thread's id = <unowned>
        object = 0x8374178

(un-named monitor @0x8374480 for object @0x8374480)
        owner thread's id = <unowned>
        object = 0x8374480

===Lines Removed===

Thread global
Raw monitor: id = <unavailable>

NLS hash table
Raw monitor: id = <unavailable>

portLibrary_j9sig_masterHandler_monitor
Raw monitor: id = <unavailable>

===Lines Removed===

(un-named monitor @0x9706b808 for object @0x9706b808)
        owner thread's id = <data unavailable>
        object = 0x9706b808

(un-named monitor @0x9706b880 for object @0x9706b880)
        owner thread's id = <data unavailable>
        object = 0x9706b880

Locked objects...
 java/lang/Class@0x9706b880 is locked by a thread with id <data unavailable>
 java/lang/Class@0x9706b808 is locked by a thread with id <data unavailable>
```

```
> info sym

        modules for address space # 0
         process id: 22650

          /usr/lib/ibm-java2-i386-60/jre/lib/i386/j9vm/libjvm.so, sections:
           0xb7e43000 - 0xb7e43000, name: "", size: 0x0
           0xb7e430d4 - 0xb7e4372c, name: ".hash", size: 0x658
           Lines Removed

          /usr/lib/ibm-java2-i386-60/jre/bin/../lib/i386/libj9thr24.so, sections:
           0xb7e38000 - 0xb7e38000, name: "", size: 0x0
           0xb7e380d4 - 0xb7e3856c, name: ".hash", size: 0x498
           Lines Removed

          Lines Removed

> info mmap
Address: 0x0   size: 0x0 (0)
Address: 0x1000   size: 0x1000 (4096)
===Lines Removed===
Address: 0xffffe478   size: 0x24 (36)
Address: 0xffffe5a8   size: 0x78 (120)
> info heap

        Runtime #1
         Heap #1:  Default@806d39c

Use "info heap *" or "info heap <heap_name>" for more information.
> info heap Default

        Runtime #1
         Heap #1:  Default@806d39c
          Size of heap: 4194304 bytes
          Occupancy :   523441 bytes  (12.47%)
          Section #1:  Heap extent at 0x96fe3000 (0x400000 bytes)
           Size:        4194304 bytes
           Shared:      false
           Executable:  false
           Read Only:   false
> hexdump 0x96fe3000 200

96fe3000: 68b20b08 0e80000c 00000000 78a70b08  |h...........x...|
96fe3010: 00000000 00000000 68b20b08 0e80060c  |........h.......|
96fe3020: 00000000 98a90b08 00000000 00000000  |................|
96fe3030: 68b20b08 0e800c0c 00000000 68ae0b08  |h...........h...|
96fe3040: 00000000 00000000 68b20b08 0e80120c  |........h.......|
96fe3050: 00000000 10af0b08 00000000 00000000  |................|
96fe3060: 68b20b08 0e80180c 00000000 d0af0b08  |h...............|
96fe3070: 00000000 00000000 68b20b08 0e801e0c  |........h.......|
96fe3080: 00000000 78b00b08 00000000 00000000  |....x...........|
96fe3090: 68b20b08 0e80240c 00000000 68b20b08  |h.....$.....h...|
96fe30a0: 00000000 00000000 68b20b08 0e802a0c  |........h.....*.|
96fe30b0: 00000000 50c80b08 00000000 00000000  |....P...........|
96fe30c0: 68b20b08 0e80300c                    |h.....0.

> +

96fe30c8: 00000000 f0c80b08 00000000 00000000  |................|
96fe30d8: 68b20b08 0e80360c 00000000 90c90b08  |h.....6.........|
96fe30e8: 00000000 00000000 68b20b08 0e803c0c  |........h.....<.|
96fe30f8: 00000000 50ca0b08 00000000 00000000  |....P...........|
96fe3108: 68b20b08 0e80420c 00000000 10cb0b08  |h.....B.........|
96fe3118: 00000000 00000000 68b20b08 0e80480c  |........h.....H.|
96fe3128: 00000000 c8cb0b08 00000000 00000000  |................|
96fe3138: 68b20b08 0e804e0c 00000000 78cc0b08  |h.....N.....x...|
96fe3148: 00000000 00000000 68b20b08 0e80540c  |........h.....T.|
96fe3158: 00000000 30cd0b08 00000000 00000000  |....0...........|
96fe3168: 68b20b08 0e805a0c 00000000 e0cd0b08  |h.....Z.........|
96fe3178: 00000000 00000000 68b20b08 0e80600c  |........h.....`.|
96fe3188: 00000000 98ce0b08                    |........

> whatis 0x96fe4000

        runtime #1
          heap #1 - name: Default@806d39c
                0x96fe4000 is within heap segment: 96fe3000 -- 973e3000
```

```
                0x96fe4000 is within an object on the heap:
                        offset 24 within java/lang/String instance @ 0x96fe3fe8

{ find command parameters are: <pattern>,<start_address>,<end_address>,<memory_boundary>,
                                        <bytes_to_print>,<matches_to_display> }
> find a,0x96fe3000,0x96ff0000,10,20,5
#0: 0x96fe45f2
#1: 0x96fe4a2a
#2: 0x96fe4ab6
#3: 0x96fe4aca
#4: 0x96fe4b56

96fe4b56: 61002e00 73007000 65006300 69006600  |a...s.p.e.c.i.f.|
96fe4b66: 69006300                             |i.c.

> findnext
#0: 0x96fe4b6a
#1: 0x96fe4c1e
#2: 0x96fe4c32
#3: 0x96fe4d54
#4: 0x96fe7a4a

96fe7a4a: 61007400 69006e00 34000000 000078d7  |a.t.i.n.4.....x.|
96fe7a5a: 0b080e80                             |....

> findnext
#0: 0x96fe8c9c
#1: 0x96fe98e0
#2: 0x96fe9b1a
#3: 0x96fea240
#4: 0x96feb5dc

96feb5dc: 61007400 69006e00 31000000 78d70b08  |a.t.i.n.1...x...|
96feb5ec: 0e807a2d                             |..z-


{ x/j can be passed an object address or a class name }
{ If passed an object address the (non-static) fields and values of the object will be printed }
> x/j 0xb1439000


        runtime #1 - version: Java(TM) 2 Runtime Environment, Standard Edition(build 2.4)
IBM J9 VM(J2RE 1.6.0 IBM J9 2.4 Linux x86-32 jvmxi3260-20070321_12042 (JIT enabled)
J9VM - 20070321_12042_lHdSMR
JIT  - dev_20070321_1807_dev
GC   - 20070319_AA)
         heap #1 - name: Default@806d39c

          java/lang/Class @ 0x96fe3000
           declared fields:
            private long vmRef = 134981496 (0x80ba778)
            private Object classLoader = null

{ If passed a class name the static fields and their values are printed }
> x/j java/lang/Float

        runtime #1 - version: Java(TM) 2 Runtime Environment, Standard Edition(build 2.4)
IBM J9 VM(J2RE 1.6.0 IBM J9 2.4 Linux x86-32 jvmxi3260-20070321_12042 (JIT enabled)
J9VM - 20070321_12042_lHdSMR
JIT  - dev_20070321_1807_dev
GC   - 20070319_AA)
         heap #1 - name: Default@806d39c

          static fields for "java/lang/Float"
            public static final float POSITIVE_INFINITY = Infinity (0x7f800000)
            public static final float NEGATIVE_INFINITY = -Infinity (0xffffffffff800000)
            public static final float NaN = NaN (0x7fc00000)
            public static final float MAX_VALUE = 3.4028235E38 (0x7f7fffff)
            public static final float MIN_NORMAL = 1.17549435E-38 (0x800000)
            public static final float MIN_VALUE = 1.4E-45 (0x1)
            public static final int MAX_EXPONENT = 127 (0x7f)
            public static final int MIN_EXPONENT = -126 (0xffffffffffffff82)
            public static final int SIZE = 32 (0x20)
            public static final Class TYPE = <object> @ 0x96fe3138
            private static final long serialVersionUID = -2671257302660747028 (0xdaedc9a2db3cf0ec)

          <no object of class "java/lang/Float" exists>

> x/d 0x96fe3000
```

```
        0x96fe3000: 134984296 <<Integer at specified address

> x/x 0x96fe3000

        0x96fe3000: 080bb268 <<Hex value of the bytes at specified address

> pwd
        /home/jammy/tests

> quit
```

jdmpview commands quick reference:

A short list of the commands you use with **jdmpview**.

The following table shows the jdmpview - quick reference:

Command	Sub-command	Description
help		Displays a list of commands or help for a specific command.
info		
	thread	Displays information about Java and native threads.
	system	Displays information about the system the core dump is from.
	class	Displays the inheritance chain and other data for a given class.
	proc	Displays threads, command line arguments, environment variables, and shared modules of current process.
	jitm	Displays JIT 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 use of defaults): passes number of items to display and unit size ('b' for byte (8 bit), 'h' for half word (16 bit), 'w' for word (32 bit), 'g' for giant word (64 bit)) to sub-command (for example x/12bd).

Command	Sub-command	Description
	J	Displays information about a particular object or all objects of a class.
	D	Displays the integer at the specified address.
	X	Displays the hex value of the bytes at the specified address.
	K	Displays the specified memory section as if it were a stack frame.
deadlock		Displays information about deadlocks if there are any set.
set 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. In most cases, the trace data is kept in a compact binary format, that can be formatted with the Java formatter that is supplied.

Tracing is enabled by default, together with a small set of trace points going to memory buffers. You can enable tracepoints at runtime by using levels, components, group names, or individual tracepoint identifiers.

This chapter describes JVM trace in:

- "What can be traced?"
- "Default tracing" on page 333
- "Where does the data go?" on page 334
- "Controlling the trace" on page 336
- "Determining the tracepoint ID of a tracepoint" on page 354
- "Application trace" on page 355
- "Using method trace" on page 358

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 337 for more information about command line options.

Auxiliary tracepoints

Auxiliary tracepoints are a special type of tracepoint that can be fired only when another tracepoint is being processed. An example of auxiliary tracepoints are the tracepoints containing the stack frame information produced by the **jstacktrace -Xtrace:trigger** command. You cannot control where auxiliary tracepoint data is sent and you cannot set triggers on auxiliary tracepoints. Auxiliary tracepoint data is sent to the same destination as the tracepoint that caused them to be generated.

Default tracing

By default, the equivalent of the following trace command line is always available in the JVM:

```
-Xtrace:maximal=all{level1},exception=j9mm{gclogger}
```

The data generated by those tracepoints is continuously captured in wrapping, per thread memory buffers. (For information about specific options, see "Detailed descriptions of trace options" on page 337.)

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 271 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 336 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.

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

To examine the trace data captured in these memory buffers, you must snap or dump the data, then format the buffers.

Snapping buffers

Under default conditions, a running JVM collects a small amount of trace data in special wraparound buffers. This data is sent to a snap trace file under certain conditions:

- An uncaught OutOfMemoryError occurs.
- An operating system signal or exception occurs.
- The com.ibm.jvm.Trace.snap() Java API is called.
- The JVMRI TraceSnap function is called.

The resulting snap trace file is placed into the current working directory, with a name in the format `Snap`*`nnnn.yyyymmdd.hhmmssth.process`*`.trc`, where *nnnn* is a sequence number reset to 0001 at JVM startup, *yyyymmdd*

is the current date, *hhmmssth* is the current time, and *process* is the process identifier. This file is in a binary format, and requires the use of the supplied trace formatter so that you can read it.

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

Extracting buffers from system dump
You can extract the buffers from a system dump core file by using the Dump Viewer.

Writing trace data to a file:

You can write trace data to a file continuously as an extension to the in-storage trace, but, instead of one buffer per thread, at least two buffers per thread are allocated, and the data is written to the file before wrapping can occur.

This allocation allows the thread to continue to run while a full trace buffer is written to disk. Depending on trace volume, buffer size, and the bandwidth of the output device, multiple buffers might be allocated to a given thread to keep pace with trace data that is being generated.

A thread is never stopped to allow trace buffers to be written. If the rate of trace data generation greatly exceeds the speed of the output device, excessive memory usage might occur and cause out-of-memory conditions. To prevent this, use the **nodynamic** option of the **buffers** trace option. For long-running trace runs, a **wrap** option is available to limit the file to a given size. It is also possible to create a sequence of files when the trace output will move back to the first file once the sequence of files are full. See the **output** option for details. You must use the trace formatter to format trace data from the file.

Because trace data is buffered, if the JVM does not exit normally, residual trace buffers might not be flushed to the file. If the JVM encounters a fatal error, the buffers can be extracted from a system dump if that is available. When a snap file is created, all available buffers are always written to it.

External tracing:

You can route trace to an agent by using JVMRI TraceRegister.

This mechanism allows a callback routine to be called immediately when any of the selected tracepoints is found without buffering the trace results. The trace data is in raw binary form. Further details can be found in the JVMRI section.

Tracing to stderr:

For lower volume or non-performance-critical tracing, the trace data can be formatted and routed to stderr immediately without buffering.

For more information, see “Using method trace” on page 358.

Trace combinations:

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

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

Controlling the trace

You have several ways by which you can control the trace.

You can control the trace in several ways by using:

- The **-Xtrace** options when launching the JVM, including trace trigger events
- A trace properties file
- com.ibm.jvm.Trace API
- JVMTI and JVMRI from an external agent

Note:

1. The specification of trace options is cumulative. Multiple **-Xtrace** options are accepted on the command line and they are processed left to right order. Each one adds to the options set by the previous one (and to the default options), as if they had all been specified in one long comma-separated list in a single option. This cumulative specification is consistent with the related **-Xdump** option processing.
2. By default, trace options equivalent to the following are enabled:
   `-Xtrace:maximal=all{level1},exception=j9mm{gclogger}`
3. To disable the defaults, use the **-Xtrace:none** option.
4. Many diagnostic tools start a JVM. When using the **IBM_JAVA_OPTIONS** environment variable trace to a file, starting a diagnostic tool might overwrite the trace data generated from your application. Use the command-line tracing options or add %d, %p or %t to the trace file name to prevent this from happening. See “Detailed descriptions of trace options” on page 337 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>
```

If you use UNIX style shells, note that unwanted shell expansion might occur because of the characters used in the trace options. To avoid unpredictable results, enclose this command-line option in quotation marks. For example:

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

For more information, see the manual for your shell.

Detailed descriptions of trace options:

The options are processed in the sequence in which they are described here.

-Xtrace command-line option syntax

```
                  ┌──,──────────────────────────────────────────────────────┐
                  ▼                                                         │
►►──-Xtrace:──────┬─properties[=<filename>]──────────────────────────────────┴──────────►◄
                  ├─buffers=<size>[,──┬─dynamic──────┬─]──────────────────────────────┤
                  │                   └─nodynamic────┘                                │
                  │                  ┌──,──────────────────────────────┐              │
                  │                  ▼                                 │              │
                  ├──┬─minimal──────┬──=──────<tracepoint_specification>──────────────┤
                  │  ├─maximal──────┤                                                 │
                  │  ├─count────────┤                                                 │
                  │  ├─print────────┤                                                 │
                  │  ├─iprint───────┤                                                 │
                  │  ├─exception────┤                                                 │
                  │  ├─external─────┤                                                 │
                  │  └─none─────────┘                                                 │
                  │            ┌──,──────────────────────────┐                        │
                  │            ▼                             │                        │
                  ├─method=──────<method_specification>──────┴────────────────────────┤
                  ├─output=<filename>[,<size>[,<generations>]]──────────────────────────┤
                  ├─exception.output=<filename>[,<size>]────────────────────────────────┤
                  ├─resume──────────────────────────────────────────────────────────────┤
                  ├─resumecount=<count>─────────────────────────────────────────────────┤
                  ├─sleeptime=nnn|aaams|bbbs────────────────────────────────────────────┤
                  ├─stackdepth=<n>──────────────────────────────────────────────────────┤
                  ├─suspend─────────────────────────────────────────────────────────────┤
                  ├─suspendcount=<count>────────────────────────────────────────────────┤
                  │            ┌──,──────────┐                                         │
                  │            ▼             │                                         │
                  └─trigger=───────<clause>──┴─────────────────────────────────────────┘
```

properties[=<filename>]:

You can use properties files to control trace. A properties file saves typing and, over time, causes a library of these files to be created. Each file is tailored to solving problems in a particular area.

This trace option allows you to specify in a file any of the other trace options, thereby reducing the length of the invocation command-line. The format of the file is a flat ASCII file that contains trace options. If *<filename>* is not specified, a default name of `IBMTRACE.properties` is searched for in the current directory. Nesting is not supported; that is, the file cannot contain a `properties` option. If any error is found when the file is accessed, JVM initialization fails with an explanatory error message and return code. All the options that are in the file are processed in the sequence in which they are stored in the file, before the next option that is obtained through the normal mechanism is processed. Therefore, a command-line property always overrides a property that is in the file.

An existing restriction means that properties that take the form **<name>=<value>** cannot be left to default if they are specified in the property file; that is, you must specify a value, for example `maximal=all`.

Another restriction means that properties files are sensitive to white space. Do not add white space before, after, or within the trace options.

You can make comments as follows:

```
// This is a comment. Note that it starts in column 1
```

Examples

- Use `IBMTRACE.properties` in the current directory:

  ```
  -Xtrace:properties
  ```

  – Use `trace.prop` in the current directory:

    ```
    -Xtrace:properties=trace.prop
    ```

  – Use `c:\trc\gc\trace.props`:

    ```
    -Xtrace:properties=c:\trc\gc\trace.props
    ```

Here is an example property file:

```
minimal=all
// maximal=j9mm
maximal=j9shr
buffers=20k
output=c:\traces\classloader.trc
print=tpnid(j9vm.23-25)
```

buffers=nnnk|nnnm[,dynamic|nodynamic]:

You can modify the size of the buffers to change how much diagnostics output is provided in a snap dump. This buffer is allocated for each thread that makes trace entries.

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 runtime and the implicit destination of the trace data.

In some cases, you must use them with other options. For example, if you specify maximal or minimal tracepoints, the trace data is put into memory buffers. If you are going to send the data to a file, you must use an output option to specify the destination filename.

minimal=[!]<*tracepoint_specification*>[,...]
maximal=[!]<*tracepoint_specification*>[,...]
count=[!]<*tracepoint_specification*>[,...]
print=[!]<*tracepoint_specification*>[,...]
iprint=[!]<*tracepoint_specification*>[,...]
exception=[!]<*tracepoint_specification*>[,...]
external=[!]<*tracepoint_specification*>[,...]
none[=<*tracepoint_specification*>[,...]]

Note that all these properties are independent of each other and can be mixed and matched in any way that you choose.

You must provide at least one tracepoint specification when using the **minimal**, **maximal**, **count**, **print**, **iprint**, **exception** and **external** options. In some older versions of the SDK the tracepoint specification defaults to 'all'.

Multiple statements of each type of trace are allowed and their effect is cumulative. To do this, you must use a trace properties file for multiple trace options of the same name.

minimal and maximal

minimal and **maximal** trace data is placed into internal trace buffers that can then be written to a snap file or written to the files that are specified in an output trace option. The **minimal** option records only the timestamp and tracepoint identifier. When the trace is formatted, missing trace data is replaced with the characters "???" in the output file. The **maximal** option specifies that all associated data is traced. If a tracepoint is activated by both trace options, **maximal** trace data is produced. Note that these types of trace are completely independent from any types that follow them. For example, if the **minimal** option is specified, it does not affect a later option such as **print**.

count

The **count** option requests that only a count of the selected tracepoints is kept. At JVM termination, all non-zero totals of tracepoints (sorted by tracepoint id) are written to a file, called utTrcCounters, in the current directory. This information is useful if you want to determine the overhead of particular tracepoints, but do not want to produce a large amount (GB) of trace data.

For example, to count the tracepoints used in the default trace configuration, use the following:

```
-Xtrace:count=all{level1},count=j9mm{gclogger}
```

print

The **print** option causes the specified tracepoints to be routed to stderr in real-time. The JVM tracepoints are formatted using `J9TraceFormat.dat`. The class library tracepoints are formatted by `TraceFormat.dat`. `J9TraceFormat.dat` and `TraceFormat.dat` are shipped in sdk/jre/lib and are automatically found by the runtime.

iprint

The **iprint** option is the same as the **print** option, but uses indenting to format the trace.

exception

When **exception** trace is enabled, the trace data is collected in internal buffers that are separate from the normal buffers. These internal buffers can then be written to a snap file or written to the file that is specified in an **exception.output** option.

The **exception** option allows low-volume tracing in buffers and files that are distinct from the higher-volume information that **minimal** and **maximal** tracing have provided. In most cases, this information is exception-type data, but you can use this option to capture any trace data that you want.

This form of tracing is channeled through a single set of buffers, as opposed to the buffer-per-thread approach for normal trace, and buffer contention might occur if high volumes of trace data are collected. A difference exists in the <*tracepoint_specification*> defaults for exception tracing; see "Tracepoint specification" on page 341.

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 333. 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>[{<type>}] or [!]<tracepoint_id>[,...]
```

where:

! is a logical not. That is, the tracepoints that are specified immediately following the **!** are turned off.

<component>
is a Java subcomponent, as detailed in Table 14. To include all Java subcomponents, specify **all**.

Table 14. Java subcomponents

Subcomponent name	Description
avl	VM AVL tree support
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 subcomponent you must also specify the `method` option.

<type> is the tracepoint type or group. The following types are supported:
- Entry
- Exit
- Event
- Exception
- Mem

- A group of tracepoints that have been specified by use of a group name. For example, nativeMethods select the group of tracepoints in MT (Method Trace) that relate to native methods. The following groups are supported:
  - compiledMethods
  - nativeMethods
  - staticMethods

<tracepoint_id>
: is the tracepoint identifier. 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 354.

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,!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 lower 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 higher for the j9mm component. For example:

```
-Xtrace:print={all,!j9trc{l5},!j9mm{l6}}
```

enables trace for all components at level 9 (the default), but disables level 6 and 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},!j9vrb,!j9trc{entry},!j9trc{exit}}
  ```

method=<method_specification>[,<method_specification>]:

Using method trace provides a complete (and potentially large) diagnosis of code paths inside your application and the system classes. Use wild cards and filtering to control method trace so that you can focus on the sections of code that interest you.

Method trace can trace:

- Method entry
- Method exit

The `methods` parameter is defined as:

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

Where:

- The delimiter between parts of the package name is a forward slash, "/".
- The **!** in the methods parameter is a NOT operator that allows you to tell the JVM not to trace the specified method or methods.

- The parentheses, **()**, define whether or not to include method parameters in the trace.
- If a method specification includes any commas, the whole specification must be enclosed in braces, for example:

  ```
  -Xtrace:methods={java/lang/*,java/util/*},print=mt
  ```
- It might be necessary to enclose your command line in quotation marks to prevent the shell intercepting and fragmenting comma-separated command lines, for example:

  ```
  "-Xtrace:methods={java/lang/*,java/util/*},print=mt"
  ```

To output all method trace information to stderr, use:

-Xtrace:print=mt,methods=*.*
: Print method trace information for all methods to stderr.

-Xtrace:iprint=mt,methods=*.*
: Print method trace information for all methods to stderr using indentation.

To output method trace information in binary format, see "output=<*filename*>[,*size*m[,<*generations*>]]" on page 347.

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 350.

Example

- Start with all tracing turned off. Each thread starts tracing when it has had three **resumethis** actions performed on it:

```
-Xtrace:resumecount=3
```

sleeptime=nnn | aaams | bbbs:

Specify how long the sleep lasts when using the sleep trigger action.

Purpose

Use this option to determine how long a sleep trigger action lasts. The default length of time is 30 seconds. If no units are specified, the default time unit is milliseconds.

Parameters

nnn
: Sleep for *nnn* milliseconds.

*aaa***ms**
: Sleep for *aaa* milliseconds.

*bbb***s**
: Sleep for *bbb* seconds.

stackdepth=<n>:

Used to limit the amount of stack frame information collected.

Purpose

Use this option to limit the maximum number of stack frames reported by the **jstacktrace** trace trigger action. All stack frames are recorded by default.

Parameters

n Record *n* stack frames

suspend:

Suspends tracing globally (for all threads and all forms of tracing) but leaves tracepoints activated.

Example

- Tracing suspended:

```
-Xtrace:suspend
```

suspendcount=<count>:

This trace option determines whether tracing is enabled for each thread.

If *<count>* is greater than zero, each thread initially has its tracing enabled and must receive *<count>* suspend this action before it stops tracing.

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

This trace option is for use with the **trigger** option. For more information, see "trigger=*<clause>*[,*<clause>*][,*<clause>*]...."

Example

- Start with all tracing turned on. Each thread stops tracing when it has had three **suspendthis** actions performed on it:

  ```
  -Xtrace:suspendcount=3
  ```

trigger=<clause>[,<clause>][,<clause>]...:

This trace option determines when various triggered trace actions occur. Supported actions include turning tracing on and off for all threads, turning tracing on or off for the current thread, or producing various dumps.

This trace option does not control what is traced. It controls only whether the information that has been selected by the other trace options is produced as normal or is blocked.

Each clause of the **trigger** option can be **tpnid{...}**, **method{...}**, or **group{...}**. You can specify multiple clauses of the same type if required, but you do not need to specify all types. The clause types are as follows:

method{*<methodspec>*[,*<entryAction>*[,*<exitAction>*[,*<delayCount>*[,*<matchcount>*]]]]}
: On entering a method that matches *<methodspec>*, the specified *<entryAction>* is run. On leaving a method that matches *<methodspec>*, the specified *<exitAction>* is run. If you specify a *<delayCount>*, the actions are performed only after a matching *<methodspec>* has been entered that many times. If you specify a *<matchCount>*, *<entryAction>* and *<exitAction>* are performed at most that many times.

group{*<groupname>*,*<action>*[,*<delayCount>*[,*<matchcount>*]]}
: On finding any active tracepoint that is defined as being in trace group *<groupname>*, for example **Entry** or **Exit**, the specified action is run. If you specify a *<delayCount>*, the action is performed only after that many active tracepoints from group *<groupname>* have been found. If you specify a *<matchCount>*, *<action>* is performed at most that many times.

tpnid{*<tpnid>*|*<tpnidRange>*,*<action>*[,*<delayCount>*[,*<matchcount>*]]}
: On finding the specified active *<tpnid>* (tracepoint ID) or a *<tpnid>* that falls inside the specified *<tpnidRange>*, the specified action is run. If you specify a *<delayCount>*, the action is performed only after the JVM finds such an active *<tpnid>* that many times. If you specify a *<matchCount>*, *<action>* is performed at most that many times.

Actions

Wherever an action must be specified, you must select from these choices:

abort
: Halt the JVM.

coredump
See **sysdump**.

heapdump
Produce a Heapdump. See "Using Heapdump" on page 306.

javadump
Produce a Javadump. See "Using Javadump" on page 286.

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 349.

resume
Resume all tracing (except for threads that are suspended by the action of the **resumecount** property and Trace.suspendThis() calls).

resumethis
Decrement the suspend count for this thread. If the suspend count is zero or below, resume tracing for this thread.

segv
Cause a segmentation violation. (Intended for use in debugging.)

sleep
Delay the current thread for a length of time controlled by the **sleeptime** option. The default is 30 seconds. See "sleeptime=nnn|aaams|bbbs" on page 349.

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 315.

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 271 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}
  ```

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
: Specifies whether to format the time stamps into human readable form. Processing can be substantially slower if you enable this option.

-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.

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

-timezone *<offset>*
: Specifies the UTC offset to apply when formatting timestamps. The offset can be positive or negative. This option also turns on timestamp formatting.

-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 taken from the top of `TraceFormat.dat`, which illustrates how this mechanism works:

```
5.1
j9bcu.0 0 1 1 N Trc_BCU_VMInitStages_Event1 "  Trace engine initialized for module j9dyn"
j9bcu.1 2 1 1 N Trc_BCU_internalDefineClass_Entry " >internalDefineClass %p"
j9bcu.2 4 1 1 N Trc_BCU_internalDefineClass_Exit " <internalDefineClass %p ->"
j9bcu.3 2 1 1 N Trc_BCU_createRomClassEndian_Entry " >createRomClassEndian searchFilename=%s"
```

The first line of the `.dat` file is an internal version number. Following the version number is a line for each tracepoint. Trace point j9bcu.0 maps to Trc_BCU_VMInitStages_Event1 for example and j9bcu.2 maps to Trc_BCU_internalDefineClass_Exit.

The format of each tracepoint entry is:

<component.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 358.

```
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 357 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 339 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. The impact is proportion to the number of methods traced. 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.
2. Use either
   - **-Xtrace:print** to route the trace to stderr.
   - **-Xtrace:maximal** and **-Xtrace:output** to route the trace to a binary compressed file using memory buffers.

Use the **methods** parameter to control what is traced. For example, to trace all methods on the String class, set **-Xtrace:methods=java/lang/String.*,print=mt**.

The **methods** parameter is formally defined as follows:

```
-Xtrace:methods=[[!]<method_spec>[,...]]
```

Where *<method_spec>* is formally defined as:

```
{*|[*]<classname>[*]}.{*|[*]<methodname>[*]}[()]
```

Note:

- The symbol "!" in the methods parameter is a NOT operator. Use this symbol to exclude methods from the trace. Use "this" with other **methods** parameters to set up a trace of the form: “trace methods of this type but not methods of that type”.
- The parentheses, (), that are in the *<method_spec>* define whether to trace method parameters.
- If a method specification includes any commas, the whole specification must be enclosed in braces:

  ```
  -Xtrace:methods={java/lang/*,java/util/*},print=mt
  ```
- You might have to enclose your command line in quotation marks. This action prevents the shell intercepting and fragmenting comma-separated command lines:

  ```
  "-Xtrace:methods={java/lang/*,java/util/*},print=mt"
  ```

Use the **print**, **maximal** and **output** options to route the trace to the required destination, where:

- **print** formats the tracepoint data while the Java application is running and writes the tracepoints to stderr.
- **maximal** saves the tracepoints into memory buffers.
- **output** writes the memory buffers to a file, in a binary compressed format.

To produce method trace that is routed to stderr, use the **print** option, specifying **mt** (method trace). For example: **-Xtrace:methods=java/lang/String.*,print=mt**.

To produce method trace that is written to a binary file from the memory buffers, use the **maximal** and **output** options. For example: **-Xtrace:methods=java/lang/String.*,maximal=mt,output=mytrace.trc**.

If you want your trace output to contain only the tracepoints you specify, use the option **-Xtrace:none** to switch off the default tracepoints. For example: **java**

-Xtrace:none -Xtrace:methods=java/lang/String.*,maximal=mt,output=mytrace.trc <class>.

Untraceable methods:

Internal Native Library (INL) native methods inside the JVM cannot be traced because they are not implemented using JNI. The list of methods that are not traceable is subject to change without notice between releases.

The INL native methods in the JVM include:

```
java.lang.Class.allocateAndFillArray
java.lang.Class.forNameImpl
java.lang.Class.getClassDepth
java.lang.Class.getClassLoaderImpl
java.lang.Class.getComponentType
java.lang.Class.getConstructorImpl
java.lang.Class.getConstructorsImpl
java.lang.Class.getDeclaredClassesImpl
java.lang.Class.getDeclaredConstructorImpl
java.lang.Class.getDeclaredConstructorsImpl
java.lang.Class.getDeclaredFieldImpl
java.lang.Class.getDeclaredFieldsImpl
java.lang.Class.getDeclaredMethodImpl
java.lang.Class.getDeclaredMethodsImpl
java.lang.Class.getDeclaringClassImpl
java.lang.Class.getEnclosingObject
java.lang.Class.getEnclosingObjectClass
java.lang.Class.getFieldImpl
java.lang.Class.getFieldsImpl
java.lang.Class.getGenericSignature
java.lang.Class.getInterfaceMethodCountImpl
java.lang.Class.getInterfaceMethodsImpl
java.lang.Class.getInterfaces
java.lang.Class.getMethodImpl
java.lang.Class.getModifiersImpl
java.lang.Class.getNameImpl
java.lang.Class.getSimpleNameImpl
java.lang.Class.getStackClass
java.lang.Class.getStackClasses
java.lang.Class.getStaticMethodCountImpl
java.lang.Class.getStaticMethodsImpl
java.lang.Class.getSuperclass
java.lang.Class.getVirtualMethodCountImpl
java.lang.Class.getVirtualMethodsImpl
java.lang.Class.isArray
java.lang.Class.isAssignableFrom
java.lang.Class.isInstance
java.lang.Class.isPrimitive
java.lang.Class.newInstanceImpl
java.lang.ClassLoader.findLoadedClassImpl
java.lang.ClassLoader.getStackClassLoader
java.lang.ClassLoader.loadLibraryWithPath
java.lang.J9VMInternals.getInitStatus
java.lang.J9VMInternals.getInitThread
java.lang.J9VMInternals.initializeImpl
java.lang.J9VMInternals.sendClassPrepareEvent
java.lang.J9VMInternals.setInitStatusImpl
java.lang.J9VMInternals.setInitThread
java.lang.J9VMInternals.verifyImpl
java.lang.J9VMInternals.getStackTrace
java.lang.Object.clone
java.lang.Object.getClass
java.lang.Object.hashCode
java.lang.Object.notify
java.lang.Object.notifyAll
```

```
java.lang.Object.wait
java.lang.ref.Finalizer.runAllFinalizersImpl
java.lang.ref.Finalizer.runFinalizationImpl
java.lang.ref.Reference.getImpl
java.lang.ref.Reference.initReferenceImpl
java.lang.reflect.AccessibleObject.checkAccessibility
java.lang.reflect.AccessibleObject.getAccessibleImpl
java.lang.reflect.AccessibleObject.getExceptionTypesImpl
java.lang.reflect.AccessibleObject.getModifiersImpl
java.lang.reflect.AccessibleObject.getParameterTypesImpl
java.lang.reflect.AccessibleObject.getSignature
java.lang.reflect.AccessibleObject.getStackClass
java.lang.reflect.AccessibleObject.initializeClass
java.lang.reflect.AccessibleObject.invokeImpl
java.lang.reflect.AccessibleObject.setAccessibleImpl
java.lang.reflect.Array.get
java.lang.reflect.Array.getBoolean
java.lang.reflect.Array.getByte
java.lang.reflect.Array.getChar
java.lang.reflect.Array.getDouble
java.lang.reflect.Array.getFloat
java.lang.reflect.Array.getInt
java.lang.reflect.Array.getLength
java.lang.reflect.Array.getLong
java.lang.reflect.Array.getShort
java.lang.reflect.Array.multiNewArrayImpl
java.lang.reflect.Array.newArrayImpl
java.lang.reflect.Array.set
java.lang.reflect.Array.setBoolean
java.lang.reflect.Array.setByte
java.lang.reflect.Array.setChar
java.lang.reflect.Array.setDouble
java.lang.reflect.Array.setFloat
java.lang.reflect.Array.setImpl
java.lang.reflect.Array.setInt
java.lang.reflect.Array.setLong
java.lang.reflect.Array.setShort
java.lang.reflect.Constructor.newInstanceImpl
java.lang.reflect.Field.getBooleanImpl
java.lang.reflect.Field.getByteImpl
java.lang.reflect.Field.getCharImpl
java.lang.reflect.Field.getDoubleImpl
java.lang.reflect.Field.getFloatImpl
java.lang.reflect.Field.getImpl
java.lang.reflect.Field.getIntImpl
java.lang.reflect.Field.getLongImpl
java.lang.reflect.Field.getModifiersImpl
java.lang.reflect.Field.getNameImpl
java.lang.reflect.Field.getShortImpl
java.lang.reflect.Field.getSignature
java.lang.reflect.Field.getTypeImpl
java.lang.reflect.Field.setBooleanImpl
java.lang.reflect.Field.setByteImpl
java.lang.reflect.Field.setCharImpl
java.lang.reflect.Field.setDoubleImpl
java.lang.reflect.Field.setFloatImpl
java.lang.reflect.Field.setImpl
java.lang.reflect.Field.setIntImpl
java.lang.reflect.Field.setLongImpl
java.lang.reflect.Field.setShortImpl
java.lang.reflect.Method.getNameImpl
java.lang.reflect.Method.getReturnTypeImpl
java.lang.String.intern
java.lang.String.isResettableJVM0
java.lang.System.arraycopy
java.lang.System.currentTimeMillis
java.lang.System.hiresClockImpl
```

```
java.lang.System.hiresFrequencyImpl
java.lang.System.identityHashCode
java.lang.System.nanoTime
java.lang.Thread.currentThread
java.lang.Thread.getStackTraceImpl
java.lang.Thread.holdsLock
java.lang.Thread.interrupted
java.lang.Thread.interruptImpl
java.lang.Thread.isInterruptedImpl
java.lang.Thread.resumeImpl
java.lang.Thread.sleep
java.lang.Thread.startImpl
java.lang.Thread.stopImpl
java.lang.Thread.suspendImpl
java.lang.Thread.yield
java.lang.Throwable.fillInStackTrace
java.security.AccessController.getAccessControlContext
java.security.AccessController.getProtectionDomains
java.security.AccessController.getProtectionDomainsImpl
org.apache.harmony.kernel.vm.VM.getStackClassLoader
org.apache.harmony.kernel.vm.VM.internImpl
```

Examples of use:

Here are some examples of method trace commands and their results.

- **Tracing entry and exit of all methods in a given class:**

  ```
  -Xtrace:methods=java/lang/String.*,print=mt
  ```

  This example traces entry and exit of all methods in the java.lang.String class. The name of the class must include the full package name, using '/' as a separator. The method name is separated from the class name by a dot '.' In this example, '*' is used to include all methods. Sample output:

  ```
  09:39:05.569 0x1a1100 mt.0 > java/lang/String.length()I Bytecode method,
  This = 8b27d8
  09:39:05.579 0x1a1100 mt.6 < java/lang/String.length()I Bytecode method
  ```

- **Tracing method input parameters:**

  ```
  -Xtrace:methods=java/lang/Thread.*(),print=mt
  ```

  This example traces all methods in the java.lang.Thread class, with the parentheses '()' indicating that the trace should also include the method call parameters. The output includes an extra line, giving the class and location of the object on which the method was called, and the values of the parameters. In this example the method call is Thread.join(long millis,int nanos), which has two parameters:

```
09:58:12.949 0x4236ce00 mt.0 > java/lang/Thread.join(JI)V Bytecode method, This = 8ffd20
09:58:12.959 0x4236ce00 mt.18 - Instance method receiver: com/ibm/tools/attach/javaSE/AttachHandler@008FFD20
arguments: ((long)1000,(int)0)
```

- **Tracing multiple methods:**

  ```
  -Xtrace:methods={java/util/HashMap.size,java/lang/String.length},print=mt
  ```

  This example traces the size method on the java.util.HashMap class and the length method on the java.lang.String class. The method specification includes the two methods separated by a comma, with the entire method specification enclosed in braces '{' and '}'. Sample output:

  ```
  10:28:19.296 0x1a1100 mt.0 > java/lang/String.length()I Bytecode method,
  This = 8c2548
  10:28:19.306 0x1a1100 mt.6 < java/lang/String.length()I Bytecode method
  ```

```
10:28:19.316 0x1a1100 mt.0 > java/util/HashMap.size()I Bytecode method,
This = 8dd7e8
10:28:19.326 0x1a1100 mt.6 < java/util/HashMap.size()I Bytecode method
```

- **Using the ! (not) operator to select tracepoints:**

  ```
  -Xtrace:methods={java/util/HashMap.*,!java/util/HashMap.put*},print
  ```

  This example traces all methods in the java.util.HashMap class except those beginning with put. Sample output:

```
10:37:42.225 0x1a1100 mt.0 > java/util/HashMap.createHashedEntry(Ljava/lang/Object;II)Ljava/util/
HashMap$Entry; Bytecode method, This = 8e09
e0
10:37:42.246 0x1a1100 mt.6 < java/util/HashMap.createHashedEntry(Ljava/lang/Object;II)Ljava/util/
HashMap$Entry; Bytecode method
10:37:42.256 0x1a1100 mt.1 > java/util/HashMap.findNonNullKeyEntry(Ljava/lang/Object;II)Ljava/util/
HashMap$Entry; Compiled method, This = 8d
d7e0
10:37:42.266 0x1a1100 mt.7 < java/util/HashMap.findNonNullKeyEntry(Ljava/lang/Object;II)Ljava/util/
HashMap$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.

- "Disabling the JIT or AOT compiler"
- "Selectively disabling the JIT or AOT compiler" on page 365
- "Locating the failing method" on page 366
- "Identifying JIT compilation failures" on page 368
- "Identifying AOT compilation failures" on page 369
- "Performance of short-running applications" on page 369
- "JVM behavior during idle periods" on page 370

Diagnosing a JIT or AOT problem

Occasionally, valid bytecodes might compile into invalid native code, causing the Java program to fail. By determining whether the JIT or AOT compiler is faulty and, if so, *where* it is faulty, you can provide valuable help to the Java service team.

About this task

This section describes how you can determine if your problem is compiler-related. This section also suggests some possible workarounds and debugging techniques for solving compiler-related problems.

Disabling the JIT or AOT compiler:

If you suspect that a problem is occurring in the JIT or AOT compiler, disable compilation to see if the problem remains. If the problem still occurs, you know that the compiler is not the cause of it.

About this task

The JIT compiler is enabled by default. The AOT compiler is also enabled, but, is not active unless shared classes have been enabled. For efficiency reasons, not all methods in a Java application are compiled. The JVM maintains a call count for

each method in the application; every time a method is called and interpreted, the call count for that method is incremented. When the count reaches the compilation threshold, the method is compiled and executed natively.

The call count mechanism spreads compilation of methods throughout the life of an application, giving higher priority to methods that are used most frequently. Some infrequently used methods might never be compiled at all. As a result, when a Java program fails, the problem might be in the JIT or AOT compiler or it might be elsewhere in the JVM.

The first step in diagnosing the failure is to determine *where* the problem is. To do this, you must first run your Java program in purely interpreted mode (that is, with the JIT and AOT compilers disabled).

Procedure

1. Remove any **-Xjit** and **-Xaot** options (and accompanying parameters) from your command line.
2. Use the **-Xint** command-line option to disable the JIT and AOT compilers. For performance reasons, do not use the **-Xint** option in a production environment.

What to do next

Running the Java program with the compilation disabled leads to one of the following:

- The failure remains. The problem is not in the JIT or AOT compiler. In some cases, the program might start failing in a different manner; nevertheless, the problem is not related to the compiler.
- The failure disappears. The problem is most likely in the JIT or AOT compiler.

  If you are not using shared classes, the JIT compiler is at fault. If you are using shared classes, you must determine which compiler is at fault by running your application with only JIT compilation enabled. Run your application with the **-Xnoaot** option instead of the **-Xint** option. This leads to one of the following:
  - The failure remains. The problem is in the JIT compiler. You can also use the **-Xnojit** instead of the **-Xnoaot** option to ensure that only the JIT compiler is at fault.
  - The failure disappears. The problem is in the AOT compiler.

Selectively disabling the JIT 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, 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.

By default, the AOT compiler only compiles methods at the “cold” optimization level. To force the AOT compiler to compile a method at a different level, use **-Xaot**:**optLevel**=<*level*>,**forceAOT**.

You specify JIT parameters as a comma-separated list, appended to the **-Xjit** option. The syntax is **-Xjit**:*<param1>*,*<param2>*=*<value>*. For example:

```
java -Xjit:verbose,optLevel=noOpt HelloWorld
```

runs the HelloWorld program, enables verbose output from the JIT, and makes the JIT generate native code without performing any optimizations. Optimization options are listed in "How the JIT compiler optimizes code" on page 57. The AOT compiler is controlled in a similar manner, using the **-Xaot** option. Use the **-Xjit** option when diagnosing JIT compiler problems, and the **-Xaot** option when 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 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, 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 (`libj9jit26.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/libj9jit26.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 289

Module=/home/test/sdk/jre/bin/libj9jit26.so
: Indicates that the error occurred in `libj9jit26.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 366). 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 366, 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 365 for more information.

JVM behavior during idle periods

You can reduce the CPU cycles consumed by an idle JVM by using the **-XsamplingExpirationTime** option to turn off the JIT sampling thread.

The JIT sampling thread profiles the running Java application to discover commonly used methods. The memory and processor usage of the sampling thread is negligible, and the frequency of profiling is automatically reduced when the JVM is idle.

In some circumstances, you might want no CPU cycles consumed by an idle JVM. To do so, specify the **-XsamplingExpirationTime**<*time*> option. Set <*time*> to the number of seconds for which you want the sampling thread to run. Use this option with care; after it is turned off, you cannot reactivate the sampling thread. Allow the sampling thread to run for long enough to identify important optimizations.

The Diagnostics Collector

The Diagnostics Collector gathers the Java diagnostics files for a problem event.

Introduction to the Diagnostics Collector

The Diagnostics Collector gathers the Java diagnostics files for a problem event.

The Java runtime produces multiple diagnostics files in response to events such as General Protection Faults, out of memory conditions or receiving unexpected operating system signals. The Diagnostics Collector runs just after the Java runtime produces diagnostics files. It searches for system dumps, Java dumps, heap dumps, Java trace dumps and the verbose GC log that match the time stamp for the problem event. If a system dump is found, then optionally the Diagnostics Collector can execute **jextract** to post-process the dump and capture extra information required to analyze system dumps. The Diagnostics Collector then produces a single .zip file containing all the diagnostics for the problem event. Steps in the collection of diagnostics are logged in a text file. At the end of the collection process, the log file is copied into the output .zip file.

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 diagnostics. 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 right 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 373 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 diagnostics from Java runtime problems

The Diagnostics Collector produces an output file for each problem event that occurs in your Java application.

When you add the command-line option **-Xdiagnosticscollector**, the Diagnostics Collector runs and produces several output .zip files. One file is produced at startup. Another file is produced for each dump event that occurs during the lifetime of the JVM. For each problem event that occurs in your Java application, one .zip file is created to hold all the diagnostics for that event. For example, an application might have multiple `OutOfMemoryErrors` but keep on running. Diagnostics Collector produces multiple .zip files, each holding the diagnostics from one `OutOfMemoryError`.

The output .zip file is written to the current working directory by default. You can specify a different location by setting the `output.dir` property in the settings file, as described in "Diagnostics Collector settings" on page 373. An output .zip file name takes the form:

```
java.<event>.<YYYYMMDD.hhmmss.pid>.zip
```

In this file name, *<event>* is one of the following names:

- abortsignal
- check
- dumpevent
- gpf
- outofmemoryerror
- usersignal
- vmstart
- vmstop

These event names refer to the event that triggered Diagnostics Collector. The name provides a hint about the type of problem that occurred. The default name is *dumpevent*, and is used when a more specific name cannot be given for any reason.

<YYYYMMDD.hhmmss.pid> is a combination of the time stamp of the dump event, and the process ID for the original Java application. *pid* is not the process ID for the Diagnostics Collector.

The Diagnostics Collector copies files that it writes to the output .zip file. It does not delete the original diagnostics information.

When the Diagnostics Collector finds a system dump for the problem event, then by default it runs **jextract** to post-process the dump and gather context information This information enables later debugging. Diagnostics Collector automates a manual step that is requested by IBM support on most platforms. You can prevent Diagnostics Collector from running **jextract** by setting the property **run.jextract** to **false** in the settings file. For more information, see "Diagnostics Collector settings" on page 373.

The Diagnostics Collector logs its actions and messages in a file named `JavaDiagnosticsCollector.<number>.log`. The log file is written to the current working directory. The log file is also stored in the output .zip file. The *<number>* component in the log file name is not significant; it is added to keep the log file names unique.

The Diagnostics Collector is a Java VM dump agent. It is run by the Java VM in response to the dump events that produce diagnostic files by default. It runs in a new Java process, using the same version of Java as the VM producing dumps.

This ensures that the tool runs the correct version of **jextract** for any system dumps produced by the original Java process.

Verifying your Java diagnostics configuration

When you enable the command-line option `-Xdiagnosticscollector`, a diagnostics configuration check runs at Java VM start. If any settings disable key Java diagnostics, a warning is reported.

The aim of the diagnostics configuration check is to avoid the situation where a problem occurs after a long time, but diagnostics are missing because they were inadvertently switched off. Diagnostic configuration check warnings are reported on `stderr` and in the Diagnostics Collector log file. A copy of the log file is stored in the `java.check.<timestamp>.<pid>.zip` output file.

If you do not see any warning messages, it means that the Diagnostics Collector has not found any settings that disable diagnostics. The Diagnostics Collector log file stored in `java.check.<timestamp>.<pid>.zip` gives the full record of settings that have been checked.

For extra thorough checking, the Diagnostics Collector can trigger a Java dump. The dump provides information about the command-line options and current Java system properties. It is worth running this check occasionally, as there are command-line options and Java system properties that can disable significant parts of the Java diagnostics. To enable the use of a Java dump for diagnostics configuration checking, set the **config.check.javacore** option to **true** in the settings file. For more information, see "Diagnostics Collector settings."

For all platforms, the diagnostics configuration check examines environment variables that can disable Java diagnostics. For reference purposes, the full list of current environment variables and their values is stored in the Diagnostics Collector log file.

Checks for operating system settings are carried out on Linux and AIX. On AIX, the settings **fullcore=true** and **pre430core=false** are checked, as well as the core and file size ulimits.

Configuring the Diagnostics Collector

The Diagnostics Collector supports various options that can be set in a properties file.

Diagnostics Collector can be configured by using options that are set in a properties file. By default, the properties file is `jre/lib/dc.properties`. If you do not have access to edit this file, or if you are working on a shared system, you can specify an alternative filename using:

```
-Xdiagnosticscollector:settings=<filename>
```

Using a settings file is optional. By default, Diagnostics Collector gathers all the main types of Java diagnostics files.

Diagnostics Collector settings:

The Diagnostics Collector has several settings that affect the way the collector works.

The settings file uses the standard Java properties format. It is a text file with one **property=value** pair on each line. Each supported property controls the Diagnostics Collector in some way. Lines that start with '#' are comments.

Parameters

file.<any_string>=<pathname>

Any property with a name starting **file.** specifies the path to a diagnostics file to collect. You can add any string as a suffix to the property name, as a reminder of which file the property refers to. You can use any number of **file.** properties, so you can tell the Diagnostics Collector to collect a list of custom diagnostic files for your environment. Using **file.** properties does not alter or prevent the collection of all the standard diagnostic files. Collection of standard diagnostic files always takes place.

Custom debugging scripts or software can be used to produce extra output files to help diagnose a problem. In this situation, the settings file is used to identify the extra debug output files for the Diagnostics Collector. The Diagnostics Collector collects the extra debug files at the point when a problem occurs. Using the Diagnostics Collector in this way means that debug files are collected immediately after the problem event, increasing the chance of capturing relevant context information.

output.dir=<output_directory_path>

The Diagnostics Collector tries to write its output .zip file to the output directory path that you specify. The path can be absolute or relative to the working directory of the Java process. If the directory does not exist, the Diagnostics Collector tries to create it. If the directory cannot be created, or the directory is not writeable, the Diagnostics Collector defaults to writing its output .zip file to the current working directory.

loglevel.file=<level>

This setting controls the amount of information written to the Diagnostics Collector log file. The default setting for this property is **config**. Valid levels are:

off No information reported.

severe Errors are reported.

warning
Report warnings in addition to information reported by **severe**.

info More detailed information in addition to that reported by **warning**.

config Configuration information reported in addition to that reported by **info**. This is the default reporting level.

fine Tracing information reported in addition to that reported by **config**.

finer Detailed tracing information reported in addition to that reported by **fine**.

finest Report even more tracing information in addition to that reported by **finer**.

all Report everything.

loglevel.console=<level>

Controls the amount of information written by the Diagnostics Collector to `stderr`. Valid values for this property are as described for loglevel.file. The default setting for this property is **warning**.

settings.id=<identifier>
Allows you to set an identifier for the settings file. If you set loglevel.file to **fine** or **lower**, the **settings.id** is recorded in the Diagnostics Collector log file as a way to check that your settings file is loaded as expected.

config.check.javacore={true|false}
Set **config.check.javacore=true** to enable a Java dump for the diagnostics configuration check at virtual machine start-up. The check means that the virtual machine start-up takes more time, but it enables the most thorough level of diagnostics configuration checking.

run.jextract=false
Set this option to prevent the Diagnostics Collector running **jextract** on detected System dumps.

Known limitations

There are some known limitations for the Diagnostics Collector.

If Java programs do not start at all on your system, for example because of a Java runtime installation problem or similar issue, the Diagnostics Collector cannot run.

The Diagnostics Collector does not respond to additional **-Xdump** settings that specify extra dump events requiring diagnostic information. For example, if you use **-Xdump** to produce dumps in response to a particular exception being thrown, the Diagnostics Collector does not collect the dumps from this event.

Garbage Collector diagnostics

This section describes how to diagnose garbage collection.

The topics that are discussed in this chapter are:

-
-
- "Verbose garbage collection logging"
- "-Xtgc tracing" on page 378

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 474 for more information).

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 `<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 474).

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 diagnostics

There are some diagnostics that are available for class-loading.

The topics that are discussed in this chapter are:

- "Class-loader command-line options"
- "Class-loader runtime diagnostics"
- "Loading from native code" on page 384

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 diagnostics

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.

For example:

```
C:\j9test>java -Dibm.cl.verbose=*HelloWorld hw.HelloWorld
```

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 'sun.boot.library.path' to load the libraries.

- If the class that makes the native call is loaded by the Extensions class loader, this loader looks in the 'java.ext.dirs' first, then 'sun.boot.library.path,' and finally the 'java.library.path', to load the libraries.
- If the class that makes the native call is loaded by the Application class loader, this loader looks in the 'sun.boot.library.path', then the 'java.library.path', to load the libraries.
- 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 diagnostics

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

For an introduction to shared classes, see "Class data sharing" on page 55.

The topics that are discussed in this chapter are:

- "Deploying shared classes"
- "Dealing with runtime bytecode modification" on page 393
- "Understanding dynamic updates" on page 396
- "Using the Java Helper API" on page 398
- "Understanding shared classes diagnostics output" on page 401
- "Debugging problems with shared classes" on page 408
- "Class sharing with OSGi ClassLoading framework" on page 412

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 "Class data sharing command-line options" on page 175.

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 392 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 398. 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.

On the AIX operating system, if you use the **–Xshareclasses:nonpersistent** option, set the environment variable MPROTECT_SHM to ON before starting the JVM. If you do not set this environment variable, the **-Xshareclasses:mprotect** option is ignored, whether you specify a value for the option or accept the default value, and no page protection occurs when you use a nonpersistent cache.

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 459 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 upper and lower limits on the amount of AOT data the JVM can store in the cache. **-Xshareclasses:noaot** prevents the JVM from storing any AOT data.

-Xscminjitdata<size> and **-Xscmaxjitdata<size>** place upper and lower 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**.

Compatibility between service releases:

Use the most recent service release of a JVM for any application.

It is not recommended for different service releases to share the same class cache concurrently. A class cache is compatible with earlier and later service releases. However, there might be small changes in the class files or the internal class file format between service releases. These changes might result in duplication of classes in the cache. For example, a cache created by a given service release can continue to be used by an updated service release, but the updated service release might add extra classes to the cache if space allows.

To reduce class duplication, if the JVM connects to a cache which was created by a different service release, it attempts to destroy the cache then re-create it. This automated housekeeping feature is designed so that when a new JVM level is used with an existing application, the cache is automatically refreshed. However, the refresh only succeeds if the cache is not in use by any other JVM. If the cache is in use, the JVM cannot refresh the cache, but uses it where possible.

If different service releases do use the same cache, the JVM disables AOT. The effect is that AOT code in the cache is ignored.

Nonpersistent shared cache cleanup:

When using UNIX System V workstations, you might need to clean up the cache files manually.

There are two ways to clean up cache file artifacts without rebooting your system:

1. Start the JVM with the **-Xsharedclasses:nonpersistent,destroy** or **-Xsharedclasses:destroyAll** command-line option.
2. Use the **ipcs** UNIX program from a command shell.

The first option cleans up all four system artifacts, which are:

- System V shared memory.
- A System V semaphore.
- A control file for the shared memory.
- A control file for the semaphore.

The second option, using **ipcs**, is required only when the JVM cannot find, and properly cleanup, the System V IPC objects allocated in the operating system. Information about the location of the System V IPC objects is held in the control files. If the control files are removed from the file system before the System V memories or semaphores are removed from the operating system, the JVM can no longer locate them. Running the **ipcs** command frees the resources from your operating system. Alternatively, you can free the resources by rebooting the system.

Manual cleanup is required when you see messages like:

```
JVMPORT021W You have opened a stale System V shared semaphore:
file:/tmp/javasharedresources/C240D2A64_semaphore_sharedcc_J9BUILD_G06 semid:
15994888
JVMPORT020W You have opened a stale System V shared memory:
file:/tmp/javasharedresources/C240D2A64_memory_sharedcc_J9BUILD_G06 shmid:1056778
```

These messages indicate that the JVM is attaching to a System V object that has no associated control file.

In response to these messages, run the following command as root:

```
ipcs -a
```

Record the System V memory and semaphore IDs using these rules:

- Record all semaphores IDs with corresponding keys having a "Most Significant Byte" (MSB) in the range 0x81 to 0x94.
- Record all memory IDs with corresponding keys having an MSB in the range 0x61 to 0x74

For each System V semaphore ID recorded, use the following command to delete the semaphore:

```
ipcrm –s <semid>
```

where `<semid>` is the System V semaphore ID.

For each System V shared memory ID recorded, use the following command to delete the shared memory:

```
ipcrm –m <shmid>
```

where `<shmid>` is the System V shared memory ID.

Dealing with runtime bytecode modification

Modifying bytecode at runtime is an increasingly popular way to engineer required function into classes. Sharing modified bytecode improves startup time, especially when the modification being used is expensive. You can safely cache modified bytecode and share it between JVMs, but there are many potential problems because of the added complexity. It is important to understand the features described in this section to avoid any potential problems.

This section contains a brief summary of the tools that can help you to share modified bytecode.

Potential problems with runtime bytecode modification:

The sharing of modified bytecode can cause potential problems.

When a class is stored in the cache, the location from which it was loaded and a time stamp indicating version information are also stored. When retrieving a class from the cache, the location from which it was loaded and the time stamp of that location are used to determine whether the class should be returned. The cache does not note whether the bytes being stored were modified before they were defined unless it is specifically told so. Do not underestimate the potential problems that this modification could introduce:

- In theory, unless all JVMs sharing the same classes are using exactly the same bytecode modification, JVMs could load incorrect bytecode from the cache. For

example, if JVM1 populates a cache with modified classes and JVM2 is not using a bytecode modification agent, but is sharing classes with the same cache, it could incorrectly load the modified classes. Likewise, if two JVMs start at the same time using different modification agents, a mix of classes could be stored and both JVMs will either throw an error or demonstrate undefined behavior.

- An important prerequisite for caching modified classes is that the modifications performed must be deterministic and final. In other words, an agent which performs a particular modification under one set of circumstances and a different modification under another set of circumstances, cannot use class caching. This is because only one version of the modified class can be cached for any given agent and once it is cached, it cannot be modified further or returned to its unmodified state.

In practice, modified bytecode can be shared safely if the following criteria are met:

- Modifications made are deterministic and final (described 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.

JVMTI redefinition and retransformation of classes:

Redefined classes are never stored in the cache. Retransformed classes are not stored in the cache by default, but caching can be enabled using the **-Xshareclasses:cacheRetransformed** option.

Redefined classes are classes containing replacement bytecode provided by a JVMTI agent at runtime, typically where classes are modified during a debugging session. Redefined classes are never stored in the cache.

Retransformed classes are classes with registered retransformation capable agents that have been called by a JVMTI agent at runtime. Unlike RedefineClasses, the RetransformClasses function allows the class definition to be changed without reference to the original bytecode. An example of retransformation is a profiling agent that adds or removes profiling calls with each retransformation. Retransformed classes are not stored in the cache by default, but caching can be enabled using the **-Xshareclasses:cacheRetransformed** option. This option will also work with modification contexts or partitions.

Further considerations for runtime bytecode modification:

There are a number of additional items that you need to be aware of when using the cache with runtime bytecode modification.

If bytecode is modified by a non-JVMTI agent and defined using the JVM's application 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 jars and compressed files, and class files stored as `.class` files on the file system, are accessed and used in different ways. The result is that the cache treats them as two different types. Updates are managed by writing file system time stamps into the cache.

Classes found or stored using a SharedClassTokenHelper cannot be maintained in this way, because Tokens are meaningless to the cache. As a direct consequence of updating the class data, AOT data is automatically updated.

Storing classes

When a classpath is stored in the cache, the Java archive and compressed files are time stamped. These time stamps are stored as part of the classpath. Directories are not time stamped. When a ROMClass is stored, if it came from a `.class` file on the file system, the `.class` file it came from is time stamped and this time stamp is stored. Directories are not time stamped because there is no guarantee that subsequent updates to a file cause an update to the directory holding the file.

If a compressed or Java archive file does not exist, the classpath containing it can still be added to the cache, but ROMClasses from this entry are not stored. If an attempt is made to add a ROMClass to the cache from a directory, but the ROMClass does not exist as a `.class` file, it is not stored in the cache.

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

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

Finding classes

When the JVM finds a class in the cache, it must make more checks than when it stores a class.

When a potential match has been found, if it is a `.class` file on the file system, the time stamps of the `.class` file and the ROMClass stored in the cache are compared. Regardless of the source of the ROMClass (`.jar` or `.class` file), every Java archive and compressed file entry in the calling classpath, up to and including the index at which the ROMClass was "found", must be checked for updates by obtaining the time stamps. Any update might mean that another version of the class being returned had already been added earlier in the classpath.

Additionally, any classpath entries that are directories might contain `.class` files that "shadow" the potential match that has been found. Class files might be created or deleted in these directories at any point. Therefore, when the classpath is walked and jars and compressed files are checked, directory entries are also checked to see whether any `.class` files have been created unexpectedly. This check involves building a string by using the classpath entry, the package names, and the class name, and then looking for the class file. This procedure is expensive if many directories are being used in class paths. Therefore, using jar files gives better shared classes performance.

Marking classes as stale

When an individual `.class` file is updated, only the class or classes stored from that `.class` file are marked "stale".

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

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

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

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

Classes in the cache that have been loaded from **c**, **d**, and **a** are marked stale. Making a single update to one jar file might cause many classes in the cache to be marked stale. To avoid massive duplication as classes are updated, stale classes can be marked as not stale, or "redeemed", if it is proved that they are not in fact stale.

Redeeming stale classes

Because classes are marked stale when a class path update occurs, many of the classes marked stale might not have updated. When a class loader stores a class, and in doing so effectively "updates" a stale class, you can "redeem" the stale class if you can prove that it has not in fact changed.

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

AOT code

A single piece of AOT code is associated with a specific method in a specific version of a class in the cache. If new classes are added to the cache as a result of a file system update, new AOT code can be generated for those classes. If a particular class becomes stale, the AOT code associated with that class also becomes stale. If a class is redeemed, the AOT code associated with that class is also redeemed. AOT code is not shared between multiple versions of the same class.

The total amount of AOT code can be limited using **-Xscmaxaot**, and cache space can be reserved for AOT code using **-Xscminaot**.

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 396). 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, the best way to deal with metadata that cannot be stored is always to load the first class from the jar. Load the class regardless of whether it exists in the cache or not. This load activity initializes the required metadata in the class loader, which can then be cached internally. When a class is then returned by findSharedClass, the function indicates where the class has been loaded from. The result is that the correct cached metadata for that class can be used.

It is not incorrect usage to use storeSharedClass to store classes that were loaded from disk, but which are already in the cache. The cache sees that the class is a duplicate of an existing class, it is not duplicated, and so the class continues to be shared. However, although it is handled correctly, a class loader that uses only storeSharedClass is less efficient than one that also makes appropriate use of findSharedClass.

Filtering

You can filter which classes are found and stored in the cache by registering an object implementing the SharedClassFilter interface with the SharedClassHelper. Before accessing the cache, the SharedClassHelper functions performs filtering using the registered SharedClassFilter object. For example, you can cache classes inside a particular package only by creating a suitable filter. To define a filter, implement the SharedClassFilter interface, which defines the following methods:

```
boolean acceptStore(String className)
boolean acceptFind(String className)
```

You must return true when you implement these functions so that a class can be found or stored in the cache. Use the supplied parameters as required. Make sure that you implement functions that do not take long to run because they are called for every find and store. Register a filter on a SharedClassHelper using the setSharingFilter(SharedClassFilter filter) function. See the Javadoc for the

SharedClassFilter interface for more information.

Applying a global filter

You can apply a SharedClassFilter to all non-bootstrap class loaders that share classes. Specify the com.ibm.oti.shared.SharedClassGlobalFilterClass system property on the command line. For example:

```
-Dcom.ibm.oti.shared.SharedClassGlobalFilterClass=<filter class name>
```

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 439, and "Removing a shared class cache" on page 441.

Understanding shared classes diagnostics output

When running in shared classes mode, a number of diagnostics tools can help you. The verbose options are used at runtime to show cache activity and you can use the printStats and printAllStats utilities to analyze the contents of a shared class cache.

This section tells you how to interpret the output.

Verbose output:

The **verbose** suboption of **-Xshareclasses** gives the most concise and simple diagnostic output on cache usage.

See "JVM command-line options" on page 459. 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 diagnostics from the Java SharedClassHelper API using the **verboseHelper** suboption.

The output is divided into information messages and error messages:

- Information messages are prefixed with:
  `Info for SharedClassHelper id <n>: <message>`
- Error messages are prefixed with:
  `Error for SharedClassHelper id <n>: <message>`

Use the Java Helper API to obtain this output; see “Using the Java Helper API” on page 398.

verboseAOT output:

VerboseAOT provides output when compiled AOT code is being found or stored in the cache.

When a cache is being populated, you might see the following:

```
Storing AOT code for ROMMethod 0x523B95C0 in shared cache... Succeeded.
```

When a populated cache is being accessed, you might see the following:

```
Finding AOT code for ROMMethod 0x524EAEB8 in shared cache... Succeeded.
```

AOT code is generated heuristically. You might not see any AOT code generated at all for a small application.

printStats utility:

The **printStats** utility prints summary information about the specified cache to the standard error output. 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

Cache contains only classes with line numbers

base address = 0x629EC000
end address = 0x639D0000
allocation pointer = 0x62AD1468

cache size = 16776768
free bytes = 12747748
ROMClass bytes = 939112
AOT bytes = 0
Reserved space for AOT bytes = -1
Maximum space for AOT bytes = -1
JIT data bytes = 2204
Reserved space for JIT data bytes = -1
Maximum space for JIT data bytes = -1
Zip cache bytes = 791856
Data bytes = 114240
Metadata bytes = 18920
Metadata % used = 1%
Class debug area size = 2162688
Class debug area used bytes = 155328
Class debug area % used = 7%

# ROMClasses = 370
# AOT Methods = 0
# Classpaths = 1
# URLs = 0
# Tokens = 0
# Zip caches = 28
```

```
# Stale classes = 0
% Stale classes = 0%

Cache is 12% full
```

In the example output, **-Xnolinenumbers= false** means the cache was created without the **-Xnolinenumbers** option being specified.

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.

 # 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 398.

 # 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 394. For more information about modification contexts, see "Modification contexts" on page 394.

 # 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 396.

% 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” on page 407.

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.

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: 0x6BFFA890 JITPROFILE: rehash Signature: ()V Address: 0x6B323738
for ROMClass java/util/Hashtable at 0x6B3227B8.
```

This output indicates that JVM 1 stored JIT data for the method rehash() in `java/util/Hashtable`. The ROMClass address is the address of the ROMClass that contains the method with JIT data. If JIT data is stale, the entry has the word `!STALE!` appended.

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 331 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 396, 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 393.

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 393, 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 182.

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 182. 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.

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" on page 413
- "JVMRI functions" on page 415
- "API calls provided by JVMRI" on page 416
- "RasInfo structure" on page 422
- "RasInfo request types" on page 423

- "Intercepting trace data" on page 423
- "Formatting" on page 424

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 455 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 455 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 AIX or Linux

To build a JVMRI agent, use the following command:

```
gcc -I<SDK>/include -o lib<agent>.so -shared <agent>.c
```

Where *<SDK>* is the directory where your SDK is installed and *<agent>* is the name of your agent.

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 422.

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 422.)

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 421.

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 below) the thread *starts* tracing (see "Tracing Java applications and the JVM" on page 331).

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 below) the thread *stops* tracing (see "Tracing Java applications and the JVM" on page 331).

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. Use it to work out which parts of a program are using the most memory or processor time.

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 top. 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 beside 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 top 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 with the sites that allocate the most memory at the top:

```
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, such as Hyades, JProfiler, and Ariadna. These are third-party tools, therefore IBM cannot make any guarantees or recommendations regarding them. IBM does provide a simple profiling agent based on this interface, HPROF. For details about its use, see "Using the HPROF Profiler" on page 425.

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 do the following tasks:

- Modify a dump.
- Modify a trace.
- Modify the logging configuration of the JVM.
- Initiate a JVM dump.
- Query the JRE native memory use.
- Find shared class caches.
- Remove shared class caches

The definitions you need when you write a JVMTI agent are provided in the header files `jvmti.h` and `ibmjvmti.h`. These files are in `sdk/include`.

The sample JVMTI agent consists of two functions:

1. Agent_OnLoad()
2. DumpStartCallback()

Agent_OnLoad()

This function is called by the JVM when the agent is loaded at JVM startup, which allows the JVMTI agent to modify JVM behavior before initialization is complete. The sample agent obtains access to the JVMTI interface using the JNI Invocation API function GetEnv(). The agent calls the APIs GetExtensionEvents() and GetExtensionFunctions() to find the JVMTI extensions supported by the JVM. These APIs provide access to the list of extensions available in the jvmtiExtensionEventInfo and jvmtiExtensionFunctionInfo structures. The sample uses an extension event and an extension function in the following way:

The sample JVMTI agent searches for the extension event VmDumpStart in the list of jvmtiExtensionEventInfo structures, using the identifier `COM_IBM_VM_DUMP_START`

provided in `ibmjvmti.h`. When the event is found, the JVMTI agent calls the JVMTI interface SetExtensionEventCallback() to enable the event, providing a function DumpStartCallback() that is called when the event is triggered.

Next, the sample JVMTI agent searches for the extension function SetVMDump in the list of jvmtiExtensionFunctionInfo structures, using the identifier `COM_IBM_SET_VM_DUMP` provided in `ibmjvmti.h`. The JVMTI agent calls the function using the jvmtiExtensionFunction pointer to set a JVM dump option `java:events=thrstart`. This option requests the JVM to trigger a javadump every time a VM thread is started.

DumpStartCallback()

This callback function issues a message when the associated extension event is called. In the sample code, DumpStartCallback() is used when the VmDumpStart event is triggered.

Compiling and running the sample JVMTI agent

Use this command to build the sample JVMTI agent on 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 271. 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 271.

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.

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 274. 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.

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 277.

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 277.

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 337.

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.

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 291. 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 459. 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 459.

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.

Using the Diagnostic Tool Framework for Java

The Diagnostic Tool Framework for Java (DTFJ) is a Java application programming interface (API) from IBM used to support the building of Java diagnostics tools. DTFJ works with data from a system dump or a Javadump.

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 318. 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 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 diagnostics tools access the following information:

- Memory locations stored in the dump (System dumps only)
- Relationships between memory locations and Java internals (System dumps only)
- Java threads running in the JVM
- Native threads held in the dump (System dumps only)
- Java classes and their 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:

- "Using the DTFJ interface"
- "DTFJ example application" on page 447

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, 5.0, 6, and 7, and Javadumps from IBM SDK for Java 6 and 7.

All DTFJ implementations support the same interface, but the DTFJ implementations supplied in Versions 5.0, 6, and 7 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 5.0, 6, and 7, and the DTFJ implementations supplied in Versions 5.0, 6, and 7 do not work with system dumps from Version 1.4.2.

Figure 5 on page 446 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 317). 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:

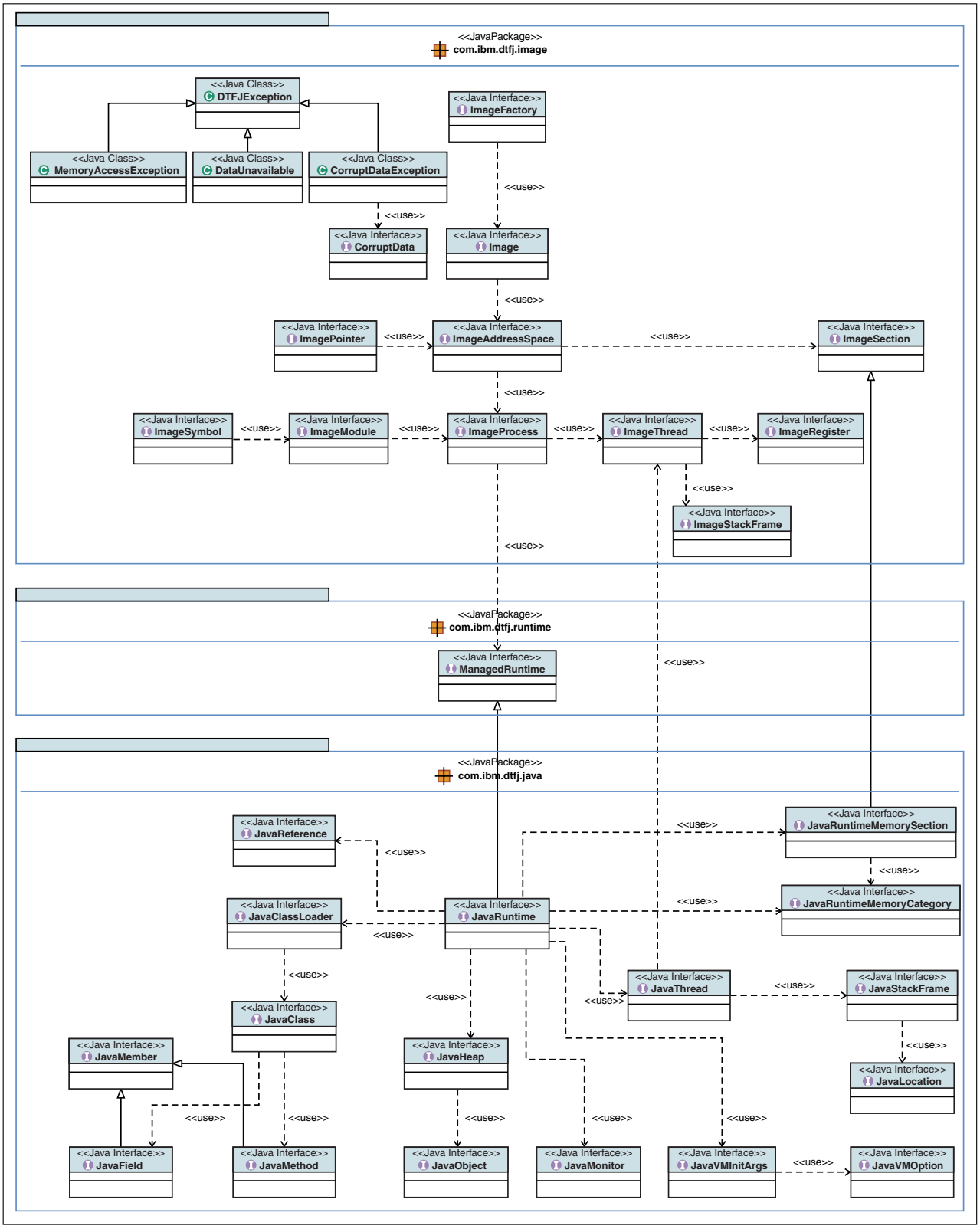


Figure 5. DTFJ interface diagram

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 291.

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 443.

In this example, the program iterates through every available Java thread and checks whether it is equal to any of the available image threads. When they are found to be equal, the program declares that it has, in this case, `"Found a match"`.

The example demonstrates:

- How to iterate down through the class hierarchy.
- How to handle CorruptData objects from the iterators.
- The use of the .equals method for testing equality between objects.

```
import java.io.File;
import java.util.Iterator;
import com.ibm.dtfj.image.CorruptData;
import com.ibm.dtfj.image.CorruptDataException;
import com.ibm.dtfj.image.DataUnavailable;
import com.ibm.dtfj.image.Image;
import com.ibm.dtfj.image.ImageAddressSpace;
import com.ibm.dtfj.image.ImageFactory;
import com.ibm.dtfj.image.ImageProcess;
import com.ibm.dtfj.java.JavaRuntime;
import com.ibm.dtfj.java.JavaThread;
import com.ibm.dtfj.image.ImageThread;
```

```
public class DTFJEX2
{
   public static void main( String[] args )
   {
      Image image = null;
      if ( args.length > 0 )
      {
         File f = new File( args[0] );
         try
         {
            Class factoryClass = Class
                  .forName( "com.ibm.dtfj.image.j9.ImageFactory" );
            ImageFactory factory = (ImageFactory) factoryClass.newInstance( );
            image = factory.getImage( f );
         }
         catch ( Exception ex )
         { /*
             * Should use the error handling as shown in DTFJEX1.
             */
            System.err.println( "Error in DTFJEX2" );
            ex.printStackTrace( System.err );
         }
      }
      else
      {
         System.err.println( "No filename specified" );
      }

      if ( null == image )
      {
         return;
      }

      MatchingThreads( image );
   }

   public static void MatchingThreads( Image image )
   {
      ImageThread imgThread = null;

      Iterator asIt = image.getAddressSpaces( );
      while ( asIt.hasNext( ) )
      {
         System.out.println( "Found ImageAddressSpace..." );

         ImageAddressSpace as = (ImageAddressSpace) asIt.next( );

         Iterator prIt = as.getProcesses( );

         while ( prIt.hasNext( ) )
         {
            System.out.println( "Found ImageProcess..." );

            ImageProcess process = (ImageProcess) prIt.next( );

            Iterator runTimesIt = process.getRuntimes( );
            while ( runTimesIt.hasNext( ) )
            {
               System.out.println( "Found Runtime..." );
               JavaRuntime javaRT = (JavaRuntime) runTimesIt.next( );

               Iterator javaThreadIt = javaRT.getThreads( );

               while ( javaThreadIt.hasNext( ) )
               {
                  Object tempObj = javaThreadIt.next( );
```

```
                /*
                 * Should use CorruptData handling for all iterators
                 */
                if ( tempObj instanceof CorruptData )
                {
                   System.out.println( "We have some corrupt data" );
                }
                else
                {
                   JavaThread javaThread = (JavaThread) tempObj;
                   System.out.println( "Found JavaThread..." );
                   try
                   {
                      imgThread = (ImageThread) javaThread.getImageThread( );

                      // Now we have a Java thread we can iterator
                      // through the image threads
                      Iterator imgThreadIt = process.getThreads( );

                      while ( imgThreadIt.hasNext( ) )
                      {
                         ImageThread imgThread2 = (ImageThread) imgThreadIt
                               .next( );
                         if ( imgThread.equals( imgThread2 ) )
                         {
                            System.out.println( "Found a match:" );
                            System.out.println( "\tjavaThread "
                                  + javaThread.getName( )
                                  + " is the same as " + imgThread2.getID( ) );
                         }
                      }
                   }
                   catch ( CorruptDataException e )
                   {
                      System.err.println( "ImageThread was corrupt: "
                            + e.getMessage( ) );
                   }
                   catch ( DataUnavailable e )
                   {
                      System.out.println( "DataUnavailable: "
                            + e.getMessage( ) );
                   }
                }
             }
          }
       }
    }
 }
```

Many DTFJ applications will follow similar models.

Using JConsole

JConsole (Java Monitoring and Management Console) is a graphical tool which allows the user to monitor and manage the behavior of Java applications.

The tool is built on top of 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 can be used to provide information about the state of the Java Virtual Machine. MXBeans are a generalized variant of MBeans. They are constructed using only a pre-defined set of data types. This means that MXBeans can be referenced and used more easily by other 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 containing the MBeans exported by the JVM. The list of available MBeans depends on the version of Java that you are using. The categories of MBean available in the `java.lang.management` package include:

- **ClassLoading**
- **Compilation**
- **GarbageCollector**

- **Memory**
- **MemoryManager**
- **MemoryPool**
- **OperatingSystem**
- **Runtime**
- **Threading**

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
: Attributes provide information about the current state. Some MBeans enable you to change the JVM options. For example, in the **Memory** MBean, you might select the `Verbose` option to enable VerboseGC logging output.

Operations
: This section shows more detailed information from the JVM. For example, in the **Threading** MBean, you see thread information that helps you to monitor deadlocked threads.

Notifications
: This section lists the notifications supported by the MBean. Applications such as JConsole receive information from the MBean by subscribing to these notifications.

Info
: This section provides details about the available notifications.

For more details on the individual platform MBeans, see the API documentation for the `java.lang.management` package at http://download.oracle.com/javase/7/docs/api/java/lang/management/package-summary.html.

GarbageCollector MBeans

Some MBeans provide garbage collection (GC) information. Expanding the **java.lang** > **GarbageCollector** item in JConsole shows you the available GC MBeans. Before Java 6 SR 7, only the J9-GC MBean was available. For Java 6 SR 7 and later, the list of MBeans depends on the GC policy in use.

For the `gencon` GC policy, two MBeans are available:

- Copy, which is for the nursery.
- MarkSweepCompact, which is for tenured memory.

For all other GC polices, only the MarkSweepCompact MBean is available.

You can use JConsole to monitor some useful GC values:

Table 15. Properties available from GarbageCollector MBeans

Property	Meaning
Name	The name of the MBean. The name is dependent on the GC policy in use. The possible values are Copy or MarkSweepCompact.

Table 15. Properties available from GarbageCollector MBeans (continued)

Property	Meaning
CollectionCount	The number of times garbage collection has occurred under the control of the garbage collector.
CollectionTime	The cumulative total of time used so far by this garbage collector, measured in milliseconds.
LastCollectionEndTime	The end time of the last collection performed by this garbage collector, measured in milliseconds since the application started.
LastCollectionStartTime	The start time of the last collection performed by this garbage collector, measured in milliseconds since the application started.
MemoryUsed	For the Copy MBean, this value is the number of nursery bytes of memory used by the heap. For the MarkSweepCompact MBean, this value is the number of tenured bytes of memory used by the heap.
TotalCompacts	The total number of times the heap has been compacted by the collector monitored by the selected bean. For the copy collector in the `gencon` GC policy, this value is always zero.
TotalMemoryFreed	The accumulated total of the memory freed by the collector, monitored by the selected bean.

OperatingSystem bean

The OperatingSystem bean shows information about the operating system in which the JVM runs, including the platform architecture, the processors, the physical memory, and system load averages. From Java 6 SR7, the following properties are included:

Table 16. Properties available from the OperatingSystem MBean

Property	Meaning
FreePhysicalMemorySize	The amount of physical memory that is currently available (free) in bytes.
ProcessVirtualMemorySize	The amount of virtual memory used (committed) by the process, in bytes. The value includes pages in physical memory and in the swap space.
ProcessPrivateMemorySize	The subset of ProcessVirtualMemoryUsed that is private to the process, in bytes. Depending on the platform, this value can include shared memory that is not in use by another process.
ProcessPhysicalMemorySize	The subset of ProcessVirtualMemoryUsed that currently occupies physical memory, in bytes. The value includes shared and private memory.

Table 16. Properties available from the OperatingSystem MBean (continued)

Property	Meaning
ProcessCpuTime	The amount of time spent running the process, in 100 nanosecond units. The value is the sum of user time and kernel (system) time.

Note: If a value of -1 is shown for a property, either that property is unavailable on the platform, or an error occurred.

Using the IBM Monitoring and Diagnostic Tools for Java - Health Center

The IBM Monitoring and Diagnostic Tools for Java - Health Center is a diagnostic tool for monitoring the status of a running Java Virtual Machine (JVM).

Information about the IBM Monitoring and Diagnostic Tools for Java - Health Center is available on developerWorks and in an InfoCenter.

Chapter 10. Reference

This part of the *Information Center* contains reference information.

The appendixes are:

- "CORBA minor codes" on page 484
- "Environment variables" on page 486
- "Command-line options"
- "Default settings for the JVM" on page 490

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"
- "General command-line options" on page 456
- "System property command-line options" on page 457
- "JVM command-line options" on page 459
- "-XX command-line options" on page 470
- "JIT and AOT command-line options" on page 471
- "Garbage Collector command-line options" on page 474

Specifying command-line options

Although the command line is the traditional way to specify command-line options, you can pass options to the JVM in other ways.

Use only single or double quotation marks for command-line options when explicitly directed to do so for the option in question. Single and double quotation marks have different meanings on different platforms, operating systems, and shells. Do not use **'-X<option>'** or **"-X<option>"**. Instead, you must use **-X<option>**. For example, do not use `'-Xmx500m'` and `"-Xmx500m"`. Write this option as `-Xmx500m`.

These precedence rules (in descending order) apply to specifying options:

1. Command line.

   For example, `java -X<option> MyClass`
2. A file containing a list of options, specified using the **–Xoptionsfile** option on the command line. For example, `java -Xoptionsfile=myoptionfile.txt MyClass`

   In the options file, specify each option on a new line; you can use the '\' character as a continuation character if you want a single option to span multiple lines. Use the '#' character to define comment lines. You cannot specify **-classpath** in an options file. Here is an example of an options file:

© Copyright IBM Corp. 2011

```
#My options file
-X<option1>
-X<option2>=\
<value1>,\
<value2>
-D<sysprop1>=<value1>
```

3. **IBM_JAVA_OPTIONS** environment variable. You can set command-line options using this environment variable. The options that you specify with this environment variable are added to the command line when a JVM starts in that environment.

   For example, `set IBM_JAVA_OPTIONS=-X<option1> -X<option2>=<value1>`

General command-line options

Use these options to print help on assert-related options, set the search path for application classes and resources, print a usage method, identify memory leaks inside the JVM, print the product version and continue, enable verbose output, and print the product version.

-cp, -classpath <directories and compressed or jar files separated by : (; on Windows)>
Sets the search path for application classes and resources. If **-classpath** and **-cp** are not used, and the **CLASSPATH** environment variable is not set, the user classpath is, by default, the current directory (`.`).

-help, -?
Prints a usage message.

-showversion
Prints product version and continues.

-verbose:<*option*>**[,**<*option*>**...]**
Enables verbose output. Separate multiple options using commas. These options are available:

class
Writes an entry to `stderr` for each class that is loaded.

dynload
Provides detailed information as each bootstrap class is loaded by the JVM:

- The class name and package
- For class files that were in a .jar file, the name and directory path of the .jar
- Details of the size of the class and the time taken to load the class

The data is written out to `stderr`. An example of the output on a Windows platform follows:

```
<Loaded java/lang/String from C:\sdk\jre\lib\vm.jar>
<Class size 17258; ROM size 21080; debug size 0>
<Read time 27368 usec; Load time 782 usec; Translate time 927 usec>
```

gc Provide verbose garbage collection information.

init
Writes information to `stderr` describing JVM initialization and termination.

jni
Writes information to `stderr` describing the JNI services called by the application and JVM.

sizes
Writes information to `stderr` describing the active memory usage settings.

stack
Writes information to `stderr` describing the Java and C stack usage for each thread.

-version
Prints product version.

System property command-line options

Use the system property command-line options to set up your system.

-D<*name*>=<*value*>
Sets a system property.

-Dcom.ibm.jsse2.renegotiate=[ALL | NONE | ABBREVIATED]
If your Java application uses JSSE for secure communication, you can disable TLS renegotiation by installing APAR IZ65239.

ALL Allow both abbreviated and unabbreviated (full) renegotiation handshakes.

NONE
Allow no renegotiation handshakes. This 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 output channel during VM operation.

-Dcom.ibm.IgnoreMalformedInput=true
Invalid UTF8 or malformed byte sequences are replaced with the standard unicode replacement character `\uFFFD`. To retain the old behavior, where invalid UTF8 or malformed byte sequences are ignored, set this system property to true.

-Dcom.ibm.tools.attach.enable=yes
Enable the Attach API for this application. The Attach API allows your application to connect to a virtual machine. Your application can then load an agent application into the virtual machine. The agent can be used to perform tasks such as monitoring the virtual machine status.

-Dcom.ibm.zipfile.closeinputstreams=true
The Java.util.zip.ZipFile class allows you to create InputStreams on files held in a compressed archive. 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 **false**.

-Dibm.jvm.bootclasspath
The value of this property is used as an additional search path, which is inserted between any value that is defined by **-Xbootclasspath/p:** and the bootclass path. The bootclass path is either the default or the one that you defined by using the **-Xbootclasspath:** option.

-Dibm.stream.nio=[true | false]
From v1.4.1 onwards, by default the IO converters are used. This option

addresses the ordering of IO and NIO converters. When this option is set to true, the NIO converters are used instead of the IO converters.

-Dil8n.vs=[true]
This system property enables awareness of Unicode Ideographic Variation Sequence (IVS) to draw characters, except in peered components. 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=[NONE | j9jit26]
Disables the Java compiler by setting to NONE. Enable JIT compilation by setting to j9jit26 (Equivalent to **–Xjit**).

-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 **-Dsun.awt.keepWorkingSetOnMinimize=true** system property stops the JVM trimming an application when it is minimized. The default behavior is to trim an application when it is minimized.

-Dsun.net.client.defaultConnectTimeout=<*value in milliseconds*>
Specifies the default value for the connect timeout for the protocol handlers used by the java.net.URLConnection class. The default value set by the protocol handlers is -1, which means that no timeout is set.

When a connection is made by an applet to a server and the server does not respond properly, the applet might seem to hang. 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=<*value in milliseconds*>
Specifies the default value for the read timeout for the protocol handlers used by the java.net.URLConnection class when reading from an input stream when a connection is established to a resource. The default value set by the protocol handlers is -1, which means that no timeout is set.

-Dsun.nio.MaxDirectMemorySize=<*value in bytes*>
Limits the native memory size for nio Direct Byte Buffer objects to the value specified.

-Dsun.rmi.transport.tcp.connectionPool=[true | any non-null value]
Enables thread pooling for the RMI ConnectionHandlers in the TCP transport layer implementation.

-Dswing.useSystemFontSettings=[false]
From v1.4.1 onwards, by default, Swing programs running with the Windows Look and Feel render with the system font set by the user instead of a

Java-defined font. As a result, fonts for v1.4.1 differ from 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.

For options that take a *<size>* parameter, suffix the number with "k" or "K" to indicate kilobytes, "m" or "M" to indicate megabytes, or "g" or "G" to indicate gigabytes.

For options that take a *<percentage>* parameter, use a number from 0 to 1. For example, 50% is 0.5.

Options that relate to the JIT are listed under "JIT and AOT command-line options" on page 471. Options that relate to the Garbage Collector are listed under "Garbage Collector command-line options" on page 474.

-X Displays help on nonstandard options.

-Xaggressive
Enables performance optimizations that are expected to be the default in future releases.

-Xargencoding
You can put Unicode escape sequences in the argument list. This option is set to off by default.

-Xbootclasspath:*<directories and compressed or Java archive files separated by : (; on* **Windows***)>*
Sets the search path for bootstrap classes and resources. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xbootclasspath/a:*<directories and compressed or Java archive files separated by : (; on* **Windows***)>*
Appends the specified directories, compressed files, or jar files to the end of the bootstrap class path. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xbootclasspath/p:*<directories and compressed or Java archive files separated by : (; on* **Windows***)>*
Adds a prefix of the specified directories, compressed files, or Java archive files to the front of the bootstrap class path. Do not deploy applications that use the **-Xbootclasspath:** or the **-Xbootclasspath/p:** option to override a class in the standard API. 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:classpath
Displays a warning message if an error is discovered in the class path; for example, a missing directory or JAR file.

-Xcheck:gc[:*<scan options>*][:*<verify options>*][:*<misc options>*]
Performs additional checks on garbage collection. By default, no checking is performed. See the output of **-Xcheck:gc:help** for more information.

-Xcheck:jni[**:help**][**:**<*option*>=<*value*>]
Performs additional checks for JNI functions. This option is equivalent to **-Xrunjnichk**. By default, no checking is performed.

-Xcheck:memory[**:**<*option*>]
Identifies memory leaks inside the JVM using strict checks that cause the JVM to exit on failure. If no option is specified, **all** is used by default. The available options are as follows:

all
Enables checking of all allocated and freed blocks on every free and allocate call. This check of the heap is the most thorough. It typically causes the JVM to exit on nearly all memory-related problems soon after they are caused. This option has the greatest affect on performance.

callsite=<*number of allocations*>

Displays callsite information every <*number of allocations*>. Deallocations are not counted. Callsite information is presented in a table with separate information for each callsite. Statistics include:

- The number and size of allocation and free requests since the last report.
- The number of the allocation request responsible for the largest allocation from each site.

Callsites are presented as `sourcefile:linenumber` for C code and assembly function name for assembler code.

Callsites that do not provide callsite information are accumulated into an "unknown" entry.

failat=<*number of allocations*>
Causes memory allocation to fail (return NULL) after <*number of allocations*>. Setting <*number of allocations*> to 13 causes the 14th allocation to return NULL. Deallocations are not counted. Use this option to ensure that JVM code reliably handles allocation failures. This option is useful for checking allocation site behavior rather than setting a specific allocation limit.

ignoreUnknownBlocks
Ignores attempts to free memory that was not allocated using the **-Xcheck:memory** tool. Instead, the **-Xcheck:memory** statistics printed out at the end of a run indicates the number of “unknown” blocks that were freed.

mprotect=<*top*|*bottom*>
Locks pages of memory on supported platforms, causing the program to stop if padding before or after the allocated block is accessed for reads or writes. An extra page is locked on each side of the block returned to the user.

If you do not request an exact multiple of one page of memory, a region on one side of your memory is not locked. The **top** and **bottom** options control which side of the memory area is locked. **top** aligns your memory blocks to the top of the page (lower address), so buffer underruns result in an application failure. **bottom** aligns your memory blocks to the bottom of the page (higher address) so buffer overruns result in an application failure.

Standard padding scans detect buffer underruns when using **top** and buffer overruns when using **bottom**.

nofree
Keeps a list of blocks already used instead of freeing memory. This list, and the list of currently allocated blocks, is checked for memory corruption on every allocation and deallocation. Use this option to detect a dangling pointer (a pointer that is "dereferenced" after its target memory is freed). This option cannot be reliably used with long-running applications (such as WebSphere Application Server), because “freed” memory is never reused or released by the JVM.

noscan
Checks for blocks that are not freed. This option has little effect on performance, but memory corruption is not detected. This option is compatible only with **subAllocator**, **callsite**, and **callsitesmall**.

quick
Enables block padding only and is used to detect basic heap corruption. Every allocated block is padded with sentinel bytes, which are verified on every allocate and free. Block padding is faster than the default of checking every block, but is not as effective.

skipto=<*number of allocations*>
Causes the program to check only on allocations that occur after <*number of allocations*>. Deallocations are not counted. Use this option to speed up JVM startup when early allocations are not causing the memory problem. The JVM performs approximately 250+ allocations during startup.

subAllocator[=<*size in MB*>]
Allocates a dedicated and contiguous region of memory for all JVM allocations. This option helps to determine if user JNI code or the JVM is responsible for memory corruption. Corruption in the JVM **subAllocator** heap suggests that the JVM is causing the problem; corruption in the user-allocated memory suggests that user code is corrupting memory. Typically, user and JVM allocated memory are interleaved.

zero
Newly allocated blocks are set to 0 instead of being filled with the `0xE7E7xxxxxxxxE7E7` pattern. Setting these blocks to 0 helps you to determine whether a callsite is expecting zeroed memory, in which case the allocation request is followed by `memset(pointer, 0, size)`.

-Xclassgc
Enables dynamic unloading of classes by the JVM. This unloading is the default behavior. To disable dynamic class unloading, use the **-Xnoclassgc** option.

-Xcompressedrefs
(64-bit only) Uses 32-bit values for references. By default, references are 64-bit.

You cannot include the **-Xcompressedrefs** option in an options file. You must specify this option on the command line, or by using the **IBM_JAVA_OPTIONS** environment variable.

-Xdbg:<*options*>
Loads debugging libraries to support the remote debugging of applications. This option is equivalent to **-Xrunjdwp**. By default, the debugging libraries are not loaded, and the VM instance is not enabled for debug.

-Xdebug
This option is deprecated. Use **-Xdbg** for debugging.

-Xdiagnosticscollector[:settings=<filename>]
Enables the Diagnostics Collector. See "The Diagnostics Collector" on page 370 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. By default, Javadump generation is enabled.

-Xdump
See "Using dump agents" on page 271 for more information.

-Xenableexplicitgc
Signals to the VM that calls to System.gc() trigger a garbage collection. This option is enabled by default.

-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. Use this flag when you are developing new code because stricter checks will become the default in future releases. By default, strict format checks are disabled.

-Xiss<*size*>
Sets the initial stack size for Java threads. By default, the stack size is set to 2 KB. Use the **-verbose:sizes** option to output the value that the VM is using.

-Xjarversion
Produces output information about the version of each jar file in the class path, the boot class path, and the extensions directory. Version information is taken from the Implementation-Version and Build-Level properties in the manifest of the jar.

-Xjni:<*suboptions*>
Sets JNI options. You can use the following suboption with the **-Xjni** option:

-Xjni:arrayCacheMax=[<*size in bytes*>**|unlimited]**
Sets the maximum size of the array cache. The default size is 8096 bytes.

-Xlinenumbers
Displays line numbers in stack traces for debugging. See also **-Xnolinenumbers**. By default, line numbers are on.

-XlockReservation
Enables an optimization that presumes a monitor is owned by the thread that last acquired it. The optimization minimizes the runtime cost of acquiring and releasing a monitor for a single thread if the monitor is rarely acquired by multiple threads.

-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 "Messages" on page 482.

-Xlp<*size*>
AIX: Requests the JVM to allocate the Java heap (the heap from which Java objects are allocated) with large (16 MB) pages, if a size is not specified. If large pages are not available, the Java heap is allocated with the next smaller page size that is supported by the system. AIX requires special configuration to enable large pages. For more information about configuring AIX support for large pages, see http://publib.boulder.ibm.com/infocenter/aix/v6r1/topic/com.ibm.aix.prftungd/doc/prftungd/large_page_ovw.htm. The SDK supports the use of large pages only to back the Java heap shared memory segments. The JVM uses shmget() with the SHM_LGPG and SHM_PIN flags to allocate large pages. The **-Xlp** option replaces the environment variable **IBM_JAVA_LARGE_PAGE_SIZE**, which is now ignored if set.

AIX, Linux, and Windows only: If a <*size*> is specified, the JVM attempts to allocate the JIT code cache memory using pages of that size. If unsuccessful, or if executable pages of that size are not supported, the JIT code cache memory is allocated using the smallest available executable page size.

-Xmso<*size*>
Sets the initial stack size for operating system threads. The default value can be determined by running the command:

```
java -verbose:sizes
```

The maximum value for the stack size varies according to platform and specific machine configuration. If you exceed the maximum value, a `java/lang/OutOfMemoryError` message is reported.

-Xnoagent
Disables support for the old JDB debugger.

-Xnoclassgc
Disables dynamic class unloading. This option disables the release of native and Java heap storage associated with Java class loaders and classes that are no longer being used by the JVM. The default behavior is as defined by **-Xclassgc**. Enabling this option is not recommended except under the direction of the IBM Java support team. The reason is the option can cause unlimited native memory growth, leading to out-of-memory errors.

-Xnolinenumbers
Disables the line numbers for debugging. See also **-Xlinenumbers**. By default, line number are on.

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. See also **-Xsigcatch**. By default, signal handling is enabled.

-Xnosigchain
Disables signal handler chaining. See also **-Xsigchain**. By default, the signal handler chaining is enabled.

-Xoptionsfile=<*file*>

Specifies a file that contains JVM options and definitions. By default, no option file is used.

The options file does not support these options:

- **-assert**
- **-fullversion**
- **-help**
- **-memorycheck**
- **-showversion**
- **-version**
- **-Xcompressedrefs**
- **-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.

<*file*> contains options that are processed as if they had been entered directly as command-line options. For example, the options file might contain:

```
-DuserString=ABC123
-Xmx256MB
```

Some options use quoted strings as parameters. Do not split quoted strings over multiple lines using the line continuation character '\'. The '¥' character is not supported as a line continuation character. For example, the following example is not valid in an options file:

```
-Xevents=vmstop,exec="cmd /c \
echo %pid has finished."
```

The following example is valid in an options file:

```
-Xevents=vmstop, \
exec="cmd /c echo %pid has finished."
```

-Xoss<*size*>
Recognized but deprecated. Use **-Xss** and **-Xmso**. Sets the maximum Java stack size for any thread. The maximum value for the stack size varies according to platform and specific machine configuration. If you exceed the maximum value, a `java/lang/OutOfMemoryError` message is reported.

-Xrdbginfo:<*host*>**:**<*port*>
Loads the remote debug information server with the specified host and port. By default, the remote debug information server is disabled.

-Xrs
Disables signal handling in the JVM. Setting **-Xrs** prevents the Java runtime from handling any internally or externally generated signals such as SIGSEGV and SIGABRT. Any signals raised are handled by the default operating system handlers. Disabling signal handling in the JVM reduces performance by approximately 2-4%, depending on the application.

Note: Linux always uses SIGUSR1.

-Xrs:sync
On UNIX systems, this option disables signal handling in the JVM for SIGSEGV, SIGFPE, SIGBUS, SIGILL, SIGTRAP, and SIGABRT signals. However, the JVM still handles the SIGQUIT and SIGTERM signals, among others. As with **-Xrs**, the use of **-Xrs:sync** reduces performance by approximately 2-4%, depending on the application.

-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 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<*size*>
Specifies cache size. This option applies only if a cache is being created and no cache of the same name exists. The default cache size is platform-dependent. You can find out the size value being used by adding **-verbose:sizes** as a command-line argument. Minimum cache size is 4 KB. Maximum cache size is platform-dependent. The size of cache that you can specify is limited by the amount of physical memory and paging space available to the system. The virtual address space of a process is shared between the shared classes cache and the Java heap. Increasing the maximum size of the Java heap reduces the size of the shared classes cache that you can create.

-Xshareclasses:<*suboptions*>

Enables class sharing. This option can take a number of suboptions, some of which are cache utilities. Cache utilities perform the required operation on the specified cache, without starting the VM. You can combine multiple suboptions, separated by commas, but the cache utilities are mutually exclusive.

You can use the following suboptions with the **-Xshareclasses** option:

cacheDir=<*directory*>
Sets the directory in which cache data is read and written. By default, <*directory*> is `/tmp/javasharedresources` on Linux, AIX, z/OS, and IBM i. You must have sufficient permissions in <*directory*>. For AIX, the directory must not be on an NFS mount for persistent caches. The JVM writes persistent cache files directly into the directory specified. Persistent cache files can be safely moved and deleted from the file system. Nonpersistent

caches are stored in shared memory and have control files that describe the location of the memory. Control files are stored in a `javasharedresources` subdirectory of the **cacheDir** specified. Do not move or delete control files in this directory. The **listAllCaches** utility, the **destroyAll** utility, and the **expire** suboption work only in the scope of a given **cacheDir**.

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 have been transformed using the JVMTI RetransformClasses function. See "JVMTI redefinition and retransformation of classes" on page 395 for more information.

destroy (Utility option)
Destroys a cache specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. A cache can be destroyed only if all JVMs using it have shut down and the user has sufficient permissions.

destroyAll (Utility option)
Tries to destroy all caches available using the specified **cacheDir** and **nonpersistent** suboptions. A cache can be destroyed only if all JVMs using it have shut down and the user has sufficient permissions.

expire=<*time in minutes*> **(Utility option)**
Destroys all caches that have been unused for the time specified before loading shared classes. This option is not a utility option because it does not cause the JVM to exit.

groupAccess
Sets operating system permissions on a new cache to allow group access to the cache. The default is user access only.

help
Lists all the command-line options.

listAllCaches (Utility option)
Lists all the compatible and incompatible caches that exist in the specified cache directory. If you do not specify **cacheDir**, the default directory is used. Summary information, such as Java version and current usage, is displayed for each cache.

mprotect=[all | default | none]
By default, the memory pages containing the cache are protected at all times, unless a specific page is being updated. This protection helps prevent accidental or deliberate corruption to the cache. The cache header

is not protected by default because this protection has a small performance cost. Specifying `all` ensures that all the cache pages are protected, including the header. Specifying `none` disables the page protection.

modified=<*modified context*>
Used when a JVMTI agent is installed that might modify bytecode at run time. If you do not specify this suboption and a bytecode modification agent is installed, classes are safely shared with an extra performance cost. The <*modified context*> is a descriptor chosen by the user; for example, *myModification1*. This option partitions the cache, so that only JVMs using context *myModification1* can share the same classes. For instance, if you run an application with a modification context and then run it again with a different modification context, all classes are stored twice in the cache. See "Dealing with runtime bytecode modification" on page 393 for more information.

name=<*name*>
Connects to a cache of a given name, creating the cache if it does not exist. This option is also used to indicate the cache that is to be modified by cache utilities; for example, **destroy**. Use the **listAllCaches** utility to show which named caches are currently available. If you do not specify a name, the default name "sharedcc_%u" is used. "%u" in the cache name inserts the current user name. You can specify "%g" in the cache name to insert the current group name.

noaot
Disables caching and loading of AOT code.

noBootclasspath
Disables the storage of classes loaded by the bootstrap class loader in the shared classes cache. Often used with the SharedClassURLFilter API to control exactly which classes are cached. See "Using the SharedClassHelper API" on page 399 for more information about shared class filtering.

none
Added to the end of a command line, disables class data sharing. This suboption overrides class sharing arguments found earlier on the command line.

nonfatal
Allows the JVM to start even if class data sharing fails. Normal behavior for the JVM is to refuse to start if class data sharing fails. If you select **nonfatal** and the shared classes cache fails to initialize, the JVM attempts to connect to the cache in read-only mode. If this attempt fails, the JVM starts without class data sharing.

nonpersistent (default for AIX and z/OS platforms)
Uses a nonpersistent cache. The cache is lost when the operating system shuts down. Nonpersistent and persistent caches can have the same name.

persistent
Uses a persistent cache. The cache is created on disk, which persists beyond operating system restarts. Nonpersistent and persistent caches can have the same name. You must always use the **persistent** suboption when running utilities such as **destroy** on a persistent cache. On AIX, you must set the **CORE_MMAP** environment variable to `yes` when using a persistent cache.

printAllStats (Utility option)
Displays detailed information about the contents of the cache specified in

the **name=**<*name*> suboption. If the name is not specified, statistics are displayed about the default cache. Every class is listed in chronological order with a reference to the location from which it was loaded. See “printAllStats utility” on page 407 for more information.

printStats[=<*data_types*>**] (Utility option)**
Displays summary information for the cache specified by the **name**, **cacheDir**, and **nonpersistent** suboptions. The most useful information displayed is how full the cache is and how many classes it contains. Stale classes are classes that have been updated on the file system and which the cache has therefore marked "stale". Stale classes are not purged from the cache and can be reused.

Specify one or more data types, separated by a plus symbol (+), to additionally see more detailed information about the cache content. Data types include AOT data, class paths and ROMMethods.See “printStats utility” on page 403 for more information.

readonly
Opens an existing cache with read-only permissions. The JVM does not create a new cache with this suboption. Opening a cache read-only prevents the JVM from making any updates to the cache. It also allows the JVM to connect to caches created by other users or groups without requiring write access. By default, this suboption is not specified.

reset
Causes a cache to be destroyed and then re-created when the JVM starts up. This option can be added to the end of a command line as **-Xshareclasses:reset**.

silent
Disables all shared class messages, including error messages. Unrecoverable error messages, which prevent the JVM from initializing, are displayed.

verbose
Gives detailed output on the cache I/O activity, listing information about classes being stored and found. Each class loader is given a unique ID (the bootstrap loader is always 0) and the output shows the class loader hierarchy at work, where class loaders must ask their parents for a class before they can load it themselves. It is typical to see many failed requests; this behavior is expected for the class loader hierarchy. The standard option **-verbose:class** also enables class sharing verbose output if class sharing is enabled.

verboseAOT
Enables verbose output when compiled AOT code is being found or stored in the cache. AOT code is generated heuristically. You might not see any AOT code generated at all for a small application. You can disable AOT caching using the **noaot** suboption. See the IBM JVM Messages Guide for a list of the messages produced: IBM JVM Messages

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 being stored and found. Each class loader is given a unique ID (the bootstrap loader is always 0) and the output shows the class loader

hierarchy at work, where class loaders must ask their parents for a class before they can load it themselves. It is typical to see many failed requests; this behavior is expected for the class loader hierarchy.

-Xsigcatch
Enables VM signal handling code. See also **-Xnosigcatch**. By default, signal handling is enabled.

-Xsigchain
Enables signal handler chaining. See also **-Xnosigchain**. By default, signal handler chaining is enabled.

-Xss<*size*>
Sets the maximum stack size for Java threads. The default is 256 KB for 32-bit JVMs and 512 KB for 64-bit JVMs. The maximum value varies according to platform and specific machine configuration. If you exceed the maximum value, a `java/lang/OutOfMemoryError` message is reported.

-Xssi<*size*>
Sets the stack size increment for Java threads. When the stack for a Java thread becomes full it is increased in size by this value until the maximum size (**-Xss**) is reached. The default is 16 KB.

-Xthr:minimizeUserCPU
Minimizes user-mode CPU usage in thread synchronization where possible. The reduction in CPU usage might be a trade-off in exchange for lower performance.

-XtlhPrefetch
Speculatively prefetches bytes in the thread local heap (TLH) ahead of the current allocation pointer during object allocation. This helps reduce the performance cost of subsequent allocations.

-Xtrace[**:help**] | [:<*option*>=<*value*>, ...]
See “Controlling the trace” on page 336 for more information.

-Xverify[:<*option*>**]**
With no parameters, enables the verifier, which is the default. Therefore, if used on its own with no parameters, for example, **-Xverify**, this option does nothing. Optional parameters are as follows:

- **all** - enable maximum verification
- **none** - disable the verifier
- **remote** - enables strict class-loading checks on remotely loaded classes

The verifier is on by default and must be enabled for all production servers. Running with the verifier off is not a supported configuration. If you encounter problems and the verifier was turned off using **-Xverify:none**, remove this option and try to reproduce the problem.

-Xzero[:<*option*>**]**

Enables reduction of the memory footprint of Java when concurrently running multiple Java invocations. **-Xzero** might not be appropriate for all types of applications because it changes the implementation of `java.util.ZipFile`, which might cause extra memory usage. **-Xzero** includes the optional parameters:

- **j9zip** - enables the j9zip 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 jars have shared caches.
- **sharezip** - puts the j9zip cache into shared memory. The j9zip cache is a map of zip entry names to file positions used to quickly find entries in the zip file. You must enable **-Xshareclasses** to avoid a warning message. When using the **sharezip** 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.

.

-XX command-line options

JVM command-line options that are specified with -XX are not stable and are not recommended for casual use.

These options are subject to change without notice.

-XXallowvmshutdown:[false|true]
This option is provided as a workaround for customer applications that cannot shut down cleanly, as described in APAR IZ59734. Customers who need this workaround should use **-XXallowvmshutdown:false**. The default option is **-XXallowvmshutdown:true** forJava 7.

-XX:MaxDirectMemorySize=<*size*>
Sets the maximum size for an nio direct buffer. By default, the maximum size is 64 MB.

-XX:-StackTraceInThrowable
This option removes stack traces from exceptions. 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 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. The **-XX:+UseCompressedOops** option is similar to specifying **-Xcompressedrefs**, which is detailed in the topic "JVM command-line options" on page 459.

-XX:-UseCompressedOops
(64-bit only) This option prevents the use of compressed references in 64-bit JVMs. The option is provided to help when porting applications from the Oracle JVM to the IBM JVM, and might not be supported in subsequent releases.

JIT and AOT command-line options

Use these JIT and AOT compiler command-line options to control code compilation.

For more information about JIT and AOT, see "JIT and AOT problem determination" on page 364.

-Xaot[:<*parameter*>=<*value*>, ...]
With no parameters, enables the AOT compiler. The AOT compiler is enabled by default but is not active unless shared classes are enabled. Using this option on its own has no effect. Use this option to control the behavior of the AOT compiler. These parameters are useful:

count=<*n*>
Where <*n*> is the number of times a method is called before it is compiled 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<*size*>
This option is used to tune performance. It 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 right size until hot interblock calls are eliminated. A reasonable starting point to tune for the optimal size is (totalNumberByteOfCompiledMethods * 1.1).

Note: Trampolines are where reflection is used to avoid inner classes.

-XcompilationThreads<*number of threads*>

This option allows you to specify the number of compilation threads used by the JIT compiler. The number of threads must be in the range 1 - 4, inclusive. Any other value prevents the JVM from starting successfully.

Setting the compilation threads to zero does not prevent the JIT from working. Instead, if you do not want the JIT to work, use the **-Xint** option.

When multiple compilation threads are used, the JIT might generate several diagnostic log files. A log file is generated for each compilation thread. The naming convention for the log file generated by the first compilation thread follows the same pattern as for IBM SDK 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

Makes the JVM use the Interpreter only, disabling the Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilers. By default, the JIT compiler is enabled. By default, the AOT compiler is enabled, but is not used by the JVM unless shared classes are also enabled.

-Xjit[:<*parameter*>**=**<*value*>**, ...]**

With no parameters, enables the JIT compiler. The JIT compiler is enabled by default, so using this option on its own has no effect. Use this option to control the behavior of the JIT compiler. Useful parameters are:

count=<*n*>

Where <*n*> is the number of times a method is called before it is compiled. For example, setting `count=0` forces the JIT compiler to compile everything on first execution.

limitFile=(<*filename*>**,** <*m*>**,** <*n*>**)**

Compile only the methods listed on lines <*m*> to <*n*> in the specified limit file. Methods not listed in the limit file and methods listed on lines outside the range are not compiled.

optlevel=[noOpt | cold | warm | hot | veryHot | scorching]

Forces the JIT compiler to compile all methods at a specific optimization level. Specifying **optlevel** might have an unexpected effect on performance, including lower overall performance.

verbose
Reports information about the JIT and AOT compiler configuration and method compilation.

-Xnoaot
Turns off the AOT compiler and disables the use of AOT-compiled code. By default, the AOT compiler is enabled but is active only when shared classes are also enabled. Using this option does not affect the JIT compiler.

-Xnojit
Turns off the JIT compiler. By default, the JIT compiler is enabled. This option does not affect the AOT compiler.

-Xquickstart
Causes the JIT compiler to run with a subset of optimizations. The effect is faster compilation times that improve startup time, but longer running applications might run slower. When the AOT compiler is active (both shared classes and AOT compilation enabled), **-Xquickstart** causes all methods to be AOT compiled. The AOT compilation improves the startup time of subsequent runs, but might reduce performance for longer running applications. **-Xquickstart** can degrade performance if it is used with long-running applications that contain hot methods. The implementation of **-Xquickstart** is subject to change in future releases. By default, **-Xquickstart** is disabled..

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<*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<*size*>
Optionally applies a maximum number of bytes in the class cache that can be used for AOT data. This option is useful if you want a certain amount of cache space guaranteed for non-AOT data. If this option is not specified, the maximum limit for AOT data is the amount of free space in the cache. The value of this option must not be smaller than the value of **-Xscminaot** and must not be larger than the value of **-Xscmx**.

-Xscmaxjitdata<*x*>
Optionally applies a maximum number of bytes in the class cache that can be used for JIT data. This option is useful if you want a certain amount of cache space guaranteed for non-JIT data. If this option is not specified, the maximum limit for JIT data is the amount of free space in the cache. The value of this option must not be smaller than the value of **-Xscminjitdata**, and must not be larger than the value of **-Xscmx**.

-Xscminaot<*size*>
Optionally applies a minimum number of bytes in the class cache to reserve for AOT data. If this option is not specified, no space is reserved for AOT data. 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<*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.

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 375 is the main diagnostic aid that is available for runtime analysis of the Garbage Collector. However, additional command-line options are available that affect the behavior of the Garbage Collector and might aid diagnostics.

For options that take a <*size*> parameter, suffix the number with "k" or "K" to indicate kilobytes, "m" or "M" to indicate megabytes, or "g" or "G" to indicate gigabytes.

For options that take a <*percentage*> parameter, use a number from 0 to 1, for example, 50% is 0.5.

-Xalwaysclassgc
Always perform dynamic class unloading checks during global collection. The default behavior is as defined by **-Xclassgc**.

-Xclassgc
Enables the collection of class objects only on class loader changes. This behavior is the default.

-Xcompactexplicitgc
Enables full compaction each time System.gc() is called.

-Xcompactgc
Compacts on all garbage collections (system and global).

The default (no compaction option specified) makes the GC compact based on a series of triggers that attempt to compact only when it is beneficial to the future performance of the JVM.

-Xconcurrentbackground<*number*>
Specifies the number of low-priority background threads attached to assist the mutator threads in concurrent mark. The default is 0 on Linux zSeries and 1 on all other platforms.

-Xconcurrentlevel<*number*>
Specifies the allocation "tax" rate. It indicates the ratio between the amount of heap allocated and the amount of heap marked. The default is 8.

-Xconcurrentslack<*size*>
Attempts to keep the specified amount of the heap space free in concurrent collectors by starting the concurrent operations earlier. This behavior can sometimes alleviate pause time problems in concurrent collectors at the cost of longer concurrent cycles, affecting total throughput. The default is 0, which is optimal for most applications.

-Xconmeter:<*soa* | *loa* | *dynamic*>
This option determines the usage of which area, LOA (Large Object Area) or SOA (Small Object Area), is metered and hence which allocations are taxed during concurrent mark. Using **-Xconmeter:soa** (the default) applies the allocation tax to allocations from the small object area (SOA). Using **-Xconmeter:loa** applies the allocation tax to allocations from the large object area (LOA). If **-Xconmeter:dynamic** is specified, the collector dynamically determines which area to meter based on which area is exhausted first, whether it is the SOA or the LOA.

-Xdisableexcessivegc
Disables the throwing of an OutOfMemory exception if excessive time is spent in the GC.

-Xdisableexplicitgc

Disables System.gc() calls.

Many applications still make an excessive number of explicit calls to System.gc() to request garbage collection. In many cases, these calls degrade performance through premature garbage collection and compactions. However, you cannot always remove the calls from the application.

The **-Xdisableexplicitgc** parameter allows the JVM to ignore these garbage collection suggestions. Typically, system administrators use this parameter in applications that show some benefit from its use.

By default, calls to System.gc() trigger a garbage collection.

-Xdisablestringconstantgc
Prevents strings in the string intern table from being collected.

-Xenableexcessivegc
If excessive time is spent in the GC, the option returns null for an allocate request and thus causes an OutOfMemory exception to be thrown. This action occurs only when the heap has been fully expanded and the time spent is making up at least 95%. This behavior is the default.

-Xenablestringconstantgc
Enables strings from the string intern table to be collected. This behavior is the default.

-Xgc:<minContractPercent | maxContractPercent | verboseFormat>
Options that change the behavior of the Garbage Collector.

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%.

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 375.
- *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:< balanced | gencon | optavgpause | optthruput >
Controls the behavior of the Garbage Collector.

The **gencon** option is the default and requests the combined use of concurrent and generational GC to help minimize the time that is spent in any garbage collection pause.

The **optavgpause** option reduces the time that is spent in these garbage collection pauses and limits the effect of increasing heap size on the length of the garbage collection pause. Use **optavgpause** if your configuration has a large heap. Enables concurrent mark.

The **optthruput** option delivers high throughput to applications, but at the cost of occasional pauses. Disables concurrent mark.

The **balanced** option uses a region-based layout for the Java heap. These regions are individually managed to reduce the maximum pause time on large heaps and increase the efficiency of garbage collection. The policy also uses a different object allocation strategy that improves application throughput on large systems that have Non-Uniform Memory Architecture (NUMA) characteristics. (x86 and POWER® platforms only) For more information about this policy, see "Balanced Garbage Collection policy" on page 39.

-Xgcthreads<*number*>
Sets the number of threads that the Garbage Collector uses for parallel operations. This total number of GC threads is composed of one application thread with the remainder being dedicated GC threads. By default, the number is set to the number of physical CPUs present, up to a maximum of 64. To set it to a different number (for example 4), use **-Xgcthreads4**. The minimum valid value is 1, which disables parallel operations, at the cost of performance. No advantage is gained if you increase the number of threads above the default setting; you are recommended not to do so.

On systems running multiple JVMs or in LPAR environments where multiple JVMs can share the same physical CPUs, you might want to restrict the number of GC threads used by each JVM. The restriction helps prevent the total number of parallel operation GC threads for all JVMs exceeding the number of physical CPUs present, when multiple JVMs perform garbage collection at the same time.

-Xgcworkpackets<*number*>
Specifies the total number of work packets available in the global collector. If not specified, the collector allocates a number of packets based on the maximum heap size.

-Xloa
Allocates a large object area (LOA). Objects are allocated in this LOA rather than the SOA. By default, the LOA is enabled for all GC policies except for subpool, where the LOA is not available.

-Xloainitial<*percentage*>
Specifies the initial percentage (between 0 and 0.95) of the current tenure space allocated to the large object area (LOA). The default value is 0.05, which is 5%.

-Xloamaximum<*percentage*>
Specifies the maximum percentage (between 0 and 0.95) of the current tenure space allocated to the large object area (LOA). The default value is 0.5, which is 50%.

-Xloaminimum<*percentage*>
Specifies the minimum percentage (between 0 and 0.95) of the current tenure

space allocated to the large object area (LOA). The LOA does not shrink below this value. The default value is 0, which is 0%.

-Xmaxe<*size*>
Sets the maximum amount by which the garbage collector expands the heap. Typically, the garbage collector expands the heap when the amount of free space falls below 30% (or by the amount specified using **-Xminf**), by the amount required to restore the free space to 30%. The **-Xmaxe** option limits the expansion to the specified value; for example `-Xmaxe10M` limits the expansion to 10 MB. By default, there is no maximum expansion size.

-Xmaxf<*percentage*>
Specifies the maximum percentage of heap that must be free after a garbage collection. If the free space exceeds this amount, the JVM tries to shrink the heap. The default value is 0.6 (60%).

-Xmaxt<*percentage*>
Specifies the maximum percentage of time to be spent in Garbage Collection. If the percentage of time rises above this value, the JVM tries to expand the heap. The default value is 13%.

-Xmca<*size*>
Sets the expansion step for the memory allocated to store the RAM portion of loaded classes. Each time more memory is required to store classes in RAM, the allocated memory is increased by this amount. By default, the expansion step is 32 KB. Use the **-verbose:sizes** option to determine the value that the VM is using. If the expansion step size you choose is too large, `OutOfMemoryError` is reported. The exact value of a "too large" expansion step size varies according to the platform and the specific machine configuration.

-Xmco<*size*>
Sets the expansion step for the memory allocated to store the ROM portion of loaded classes. Each time more memory is required to store classes in ROM, the allocated memory is increased by this amount. By default, the expansion step is 128 KB. Use the **-verbose:sizes** option to determine the value that the VM is using. If the expansion step size you choose is too large, `OutOfMemoryError` is reported. The exact value of a "too large" expansion step size varies according to the platform and the specific machine configuration.

-Xmine<*size*>
Sets the minimum amount by which the Garbage Collector expands the heap. Typically, the garbage collector expands the heap by the amount required to restore the free space to 30% (or the amount specified using **-Xminf**). The **-Xmine** option sets the expansion to be at least the specified value; for example, `-Xmine50M` sets the expansion size to a minimum of 50 MB. By default, the minimum expansion size is 1 MB.

-Xminf<*percentage*>
Specifies the minimum percentage of heap to be left free after a garbage collection. If the free space falls below this amount, the JVM attempts to expand the heap. The default value is 30%.

-Xmint<*percentage*>
Specifies the minimum percentage of time to spend in Garbage Collection. If the percentage of time drops below this value, the JVM tries to shrink the heap. The default value is 5%.

-Xmn<*size*>
Sets the initial and maximum size of the new area to the specified value when using **-Xgcpolicy:gencon**. Equivalent to setting both **-Xmns** and **-Xmnx**. If you

set either **-Xmns** or **-Xmnx**, you cannot set **-Xmn**. If you try to set **-Xmn** with either **-Xmns** or **-Xmnx**, the VM does not start, returning an error. By default, **-Xmn** is not set. If the scavenger is disabled, this option is ignored.

-Xmns<*size*>
Sets the initial size of the new area to the specified value when using **-Xgcpolicy:gencon**. By default, this option is set to 25% of the value of the **-Xms** option. This option returns an error if you try to use it with **-Xmn**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using. If the scavenger is disabled, this option is ignored.

-Xmnx<*size*>
Sets the maximum size of the new area to the specified value when using **-Xgcpolicy:gencon**. By default, this option is set to 25% of the value of the **-Xmx** option. This option returns an error if you try to use it with **-Xmn**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using. If the scavenger is disabled, this option is ignored.

-Xmo<*size*>
Sets the initial and maximum size of the old (tenured) heap to the specified value when using **-Xgcpolicy:gencon**. Equivalent to setting both **-Xmos** and **-Xmox**. If you set either **-Xmos** or **-Xmox**, you cannot set **-Xmo**. If you try to set **-Xmo** with either **-Xmos** or **-Xmox**, the VM does not start, returning an error. By default, **-Xmo** is not set.

-Xmoi<*size*>
Sets the amount the Java heap is incremented when using **-Xgcpolicy:gencon**. If set to zero, no expansion is allowed. By default, the increment size is calculated on the expansion size, set by **-Xmine** and **-Xminf**.

-Xmos<*size*>
Sets the initial size of the old (tenure) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is set to 75% of the value of the **-Xms** option. This option returns an error if you try to use it with **-Xmo**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

-Xmox<*size*>
Sets the maximum size of the old (tenure) heap to the specified value when using **-Xgcpolicy:gencon**. By default, this option is set to the same value as the **-Xmx** option. This option returns an error if you try to use it with **-Xmo**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

-Xmr<*size*>
Sets the size of the Garbage Collection "remembered set". This set is a list of objects in the old (tenured) heap that have references to objects in the new area. By default, this option is set to 16 K.

-Xmrx<*size*>
Sets the remembered maximum size setting.

-Xms<*size*>
Sets the initial Java heap size. You can also use the **-Xmo** option. The minimum size is 1 MB.

If scavenger is enabled, **-Xms** >= **-Xmn** + **-Xmo**.

If scavenger is disabled, **-Xms** >= **-Xmo**.

-Xmx<*size*>
Sets the maximum memory size (**-Xmx** >= **-Xms**)

Examples of the use of **-Xms** and **-Xmx**:

-Xms2m -Xmx64m
Heap starts at 2 MB and grows to a maximum of 64 MB.

-Xms100m -Xmx100m
Heap starts at 100 MB and never grows.

-Xms20m -Xmx1024m
Heap starts at 20 MB and grows to a maximum of 1 GB.

-Xms50m
Heap starts at 50 MB and grows to the default maximum.

-Xmx256m
Heap starts at default initial value and grows to a maximum of 256 MB.

-Xnoclassgc
Disables class garbage collection. This option switches off garbage collection of storage associated with Java classes that are no longer being used by the JVM. The default behavior is as defined by **-Xclassgc**. By default, class garbage collection is performed.

-Xnocompactexplicitgc
Disables compaction on System.gc() calls. Compaction takes place on global garbage collections if you specify **-Xcompactgc** or if compaction triggers are met. By default, compaction is enabled on calls to System.gc().

-Xnocompactgc
Disables compaction on all garbage collections (system or global). By default, compaction is enabled.

-Xnoloa
Prevents allocation of a large object area; all objects are allocated in the SOA. See also **-Xloa**.

-Xnuma:none
Use this option to turn off Non-Uniform Memory Architecture (NUMA) awareness when using the balanced garbage collection policy. The option is enabled by default. However, for workloads that do most of their work in one thread, or workloads that maintain a full heap, turning off NUMA awareness can improve performance.

-Xsoftmx<*size*> **(AIX only)**
This option sets the initial maximum size of the Java heap. Use the **-Xmx** option to set the maximum heap size. Use the AIX DLPAR API in your application to alter the heap size limit between **-Xms** and **-Xmx** at run time. By default, this option is set to the same value as **-Xmx**.

-Xsoftrefthreshold<*number*>
Sets the value used by the GC to determine the number of GCs after which a soft reference is cleared if its referent has not been marked. The default is 32, meaning that the soft reference is cleared after 32 * (percentage of free heap space) GC cycles where its referent was not marked. For example, if **-Xsoftrefthreshold** is set to 32, and the heap is 50% free, soft references are cleared after 16 GC cycles.

-Xtgc:<*arguments*>
Provides GC tracing options, where <*arguments*> is a comma-separated list containing one or more of the following arguments:

backtrace
Before a garbage collection, a single line is printed containing the name of

the master thread for garbage collection, as well as the value of the osThread slot in the J9VMThread structure.

compaction
Prints extra information showing the relative time spent by threads in the “move” and “fixup” phases of compaction

concurrent
Prints extra information showing the activity of the concurrent mark background thread

dump
Prints a line of output for every free chunk of memory in the system, including "dark matter" (free chunks that are not on the free list for some reason, typically because they are too small). Each line contains the base address and the size in bytes of the chunk. If the chunk is followed in the heap by an object, the size and class name of the object is also printed. This argument has a similar effect to the **terse** argument.

freeList
Before a garbage collection, prints information about the free list and allocation statistics since the last GC. Prints the number of items on the free list, including "deferred" entries (with the scavenger, the unused space is a deferred free list entry). For TLH and non-TLH allocations, prints the total number of allocations, the average allocation size, and the total number of bytes discarded during allocation. For non-TLH allocations, also included is the average number of entries that were searched before a sufficiently large entry was found.

parallel
Produces statistics on the activity of the parallel threads during the mark and sweep phases of a global GC.

scavenger
Prints extra information after each scavenger collection. A histogram is produced showing the number of instances of each class, and their relative ages, present in the survivor space. The information is obtained by performing a linear walk-through of the space.

terse
Dumps the contents of the entire heap before and after a garbage collection. For each object or free chunk in the heap, a line of trace output is produced. Each line contains the base address, "a" if it is an allocated object, and "f" if it is a free chunk, the size of the chunk in bytes, and, if it is an object, its class name.

-Xverbosegclog[:<*file*>[,<X>,<Y>]]

Causes **-verbose:gc** output to be written to the specified file. If the file cannot be found, **-verbose:gc** tries to create the file, and then continues as normal if it is successful. If it cannot create the file (for example, if an invalid filename is passed into the command), it redirects the output to stderr.

If you specify <*X*> and <*Y*> the **-verbose:gc** output is redirected to X files, each containing Y GC cycles.

The dump agent tokens can be used in the filename. See “Dump agent tokens” on page 282 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

- **-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 474.

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)
- **-Xsoftmx**<*size*>

A detailed description of these command-line options can be found in "Garbage Collector command-line options" on page 474.

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:
  - `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 `JVMDUMP006I`, `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 459.

Finding logged messages

Logged messages can be found in different locations, according to platform.

Finding AIX messages

On AIX, messages are logged by the syslog daemon. By default, the syslog daemon does not run, therefore you must start it manually. You can redirect messages from the syslog daemon to the AIX error log facility by performing the following configuration steps:

1. Set up a redirect in the file `syslog.conf` so that syslog messages are sent to the error log, by adding the following line:

   ```
   user.debug errlog
   ```

2. If **syslogd** is already running, reload the updated configuration by running the following command:

   ```
   refresh -s syslogd
   ```

3. The updated configuration is used each time **syslogd** starts.
4. Use the AIX **errpt** command or the System Management Interface Tool (SMIT) to read the messages sent to the error log.

If you do not enable syslog to err log redirection, logged messages go in to the default syslog file. If **syslogd** is not running, logged messages are lost. For more

information about AIX logging, see: General Programming Concepts: Writing and Debugging Programs

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 250 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, a retry or restart of 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 "does not work".

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" on page 487
- "Separating values in a list" on page 487
- "JVM environment settings" on page 487

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 336, 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**.

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.

PLUGIN_HOME=<*path*>
Set this variable to define the path to the Java plug-in.

This variable is for AIX 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 271.

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.

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.

IBM_JAVACOREDIR=<*directory*>
This variable specifies an alternative location for Javadump files; for example, on Linux `IBM_JAVACOREDIR=/dumps`

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 284.

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.

Diagnostics options

The following list describes the diagnostics 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.

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
Javadumps	Enabled	ec
Javadumps on out of memory	Enabled	ec
Heapdumps	Disabled	ec
Heapdumps on out of memory	Enabled	ec
Sysdumps	Enabled	ec
Where dump files are produced	Current® directory	ec
Verbose output	Disabled	c
Boot classpath search	Disabled	c
JNI checks	Disabled	c
Remote debugging	Disabled	c
Strict 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

JVM setting	Default	Setting affected by
Class data sharing	Disabled	c
Accessibility support	Enabled	e
JIT compiler	Enabled	ec
AOT compiler (AOT is not used by the JVM unless shared classes are also enabled)	Enabled	c
JIT debug options	Disabled	c
Java2D max size of fonts with algorithmic bold	14 point	e
Java2D use rendered bitmaps in scalable fonts	Enabled	e
Java2D freetype font rasterizing	Enabled	e
Java2D use AWT fonts	Disabled	e

JVM setting	AIX	IBM i	Linux	Windows	z/OS	Setting affected by
Default locale	None	None	None	N/A	None	e
Time to wait before starting plug-in	N/A	N/A	Zero	N/A	N/A	e
Temporary directory	`/tmp`	`/tmp`	`/tmp`	`c:\temp`	`/tmp`	e
Plug-in redirection	None	None	None	N/A	None	e
IM switching	Disabled	Disabled	Disabled	N/A	Disabled	e
IM modifiers	Disabled	Disabled	Disabled	N/A	Disabled	e
Thread model	N/A	N/A	N/A	N/A	Native	e
Initial stack size for Java Threads 32-bit. Use: **-Xiss<size>**	2 KB	2 KB	2 KB	2 KB	2 KB	c
Maximum stack size for Java Threads 32-bit. Use: **-Xss<size>**	256 KB	256 KB	256 KB	256 KB	256 KB	c
Stack size for OS Threads 32-bit. Use **-Xmso<size>**	256 KB	256 KB	256 KB	32 KB	256 KB	c
Initial stack size for Java Threads 64-bit. Use: **-Xiss<size>**	2 KB	N/A	2 KB	2 KB	2 KB	c
Maximum stack size for Java Threads 64-bit. Use: **-Xss<size>**	512 KB	N/A	512 KB	512 KB	512 KB	c
Stack size for OS Threads 64-bit. Use **-Xmso<size>**	256 KB	N/A	256 KB	256 KB	256 KB	c
Initial heap size. Use **-Xms<size>**	4 MB	4 MB	4 MB	4 MB	4 MB	c
Maximum Java heap size. Use **-Xmx<size>**	Half the available memory with a minimum of 16 MB and a maximum of 512 MB	2 GB	Half the available memory with a minimum of 16 MB and a maximum of 512 MB	Half the real memory with a minimum of 16 MB and a maximum of 2 GB	Half the available memory with a minimum of 16 MB and a maximum of 512 MB	c

"Available memory" is defined as being the smallest of two values:

- The real or “physical” memory.
- The **RLIMIT_AS** value.

Known issues and limitations

Problems that you might encounter with the SDK and Runtime Environment for AIX in specific system environments, or configurations.

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
   ```

These known issues and limitations also apply to earlier releases of the SDK and JRE:

Web Start and IBM SDK for Java, v1.3 applications

The IBM 64-bit SDK for AIX, v7 Web Start does not support launching IBM SDK for Java 1.3 applications.

CapsLock in Chinese and Japanese locales

Note: This limitation applies to Chinese and Japanese locales only.
The CapsLock key does not work properly. On Chinese locales, you cannot enter uppercase characters by using the keyboard. To resolve the problem, see APAR IY98336.

IBM Java Desktop API

If one or more GNOME libraries are not available, the IBM Java Desktop API might not work.

Switching input methods

You must close the candidate window and commit pre-edited strings before you switch the Input Method (IM) by using the IM selection menu. If you open the IM selection menu without either closing the candidate window or committing a pre-edited string, cancel the menu, then close the candidate window, and finally commit the pre-edited string. You can then try to switch the IM again.

Displaying DBCS characters in a JFrame

DBCS characters might not display correctly in the title of a JFrame. To avoid this problem, set the language in the terminal login screen instead of in a prompt after you have logged in.

Unicode Shift_JIS code page alias

Note: This limitation applies to Japanese users only.
The Unicode code page alias "\u30b7\u30d5\u30c8\u7b26\u53f7\u5316\u8868\u73fe" for Shift_JIS has been removed. If you use this code page in your applications, replace it with Shift_JIS.

-Xshareclasses:<options>

Shared class caches and control files are not compatible between IBM SDK for Java v7 and IBM SDK for Java v6 SR3 or previous releases.

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.

Change in default page size increases memory usage

The Java heap is allocated with 64K pages by default, instead of 4K pages. This change improves application throughput and startup performance. However, the change might cause a slight increase in the amount of memory used by your application. If memory usage is critical to your application, follow both these steps to revert to 4K pages:

1. Set the environment variable **`LDR_CNTRL=TEXTPSIZE=4K@DATAPSIZE=4K@STACKPSIZE=4K`**. For more information about this environment variable, see “Working with the **`LDR_CNTRL`** environment variable” on page 159.
2. Use the Java option **`-Xlp4K`**. For more information about the **`-Xlp`** option, see “JVM command-line options” on page 459.

The change has no affect if you are already using the **`-Xlp`** option to allocate the Java heap with large pages.

Support for virtualization software

The IBM SDK for Java is supported on a number of virtualized environments.

The virtualization capabilities of the AIX POWER hypervisor, provided as part of the IBM POWER platform, have been fully tested with all SDK supported releases of AIX.

The virtualization capabilities of the Processor Resource/System Manager (PR/SM™) provided as part of the IBM z-31 and IBM z-64 platforms have been fully tested with all SDK supported releases of z/OS.

In addition, the following virtualization software has been tested for the IBM SDK for Java:

Table 17. Virtualization software tested for the IBM SDK for Java

Virtualization software	**SDK release tested**
z/VM® v6.1	GA
VMWare ESXi Server 4	GA
Kernel-based Virtual Machine (KVM) on Red Hat Enterprise Virtualization V5	GA
Microsoft Hyper-V R2	GA

Note: SDK release tested indicates the earliest release of the SDK tested.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1758
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web

© Copyright IBM Corp. 2011

sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

JIMMAIL@uk.ibm.com

[Hursley Java Technology Center (JTC) contact]

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided “AS IS”, without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at Copyright and trademark information.

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