IBM SDK, Java Technology Edition
Version 7 Release 1

AIX User Guide

IBM
Note

Before you use this information and the product it supports, read the information in “Notices” on page 623.

Copyright information

This edition of the user guide applies to the IBM SDK, Java Technology Edition, Version 7 Release 1, for all supported AIX architectures, and to all subsequent releases, modifications, and fix packs, until otherwise indicated in new editions.

This guide applies to the following programs:

- IBM 64-bit SDK for AIX, Java Technology Edition, Version 7 Release 1

Portions © Copyright 1997, 2018, Oracle and/or its affiliates.

© Copyright IBM Corporation 2011, 2018.

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 ................................................................ 1
   IBM Software Developers Kit (SDK) ......................... 2
   Runtime Environment ............................................ 4
   J9 Virtual Machine (JVM) .......................................... 6

What's new ............................................................... 6
   First release .......................................................... 7
   Service refresh 1 ..................................................... 9
   Service refresh 2 ................................................... 10
   Service refresh 3 ................................................... 13
   Service refresh 4 ................................................... 15

Packed object evaluation technology ......................... 16
   Packed objects .................................................... 16

Conventions and terminology .................................. 62
   Other sources of information ................................ 64
   Accessibility ........................................................ 64

**Chapter 2. Understanding the components** ................. 65
   The building blocks of the IBM Virtual Machine for Java .............................................. 65
      Java application stack ........................................ 66
      Components of the IBM Virtual Machine for Java ...................................................... 67
   Memory management ................................................ 69
      Overview of memory management ......................... 69
      Allocation ......................................................... 71
      Detailed description of global garbage collection ...................................................... 74
      Generational Concurrent Garbage Collector ................................................................. 83
      Balanced Garbage Collection policy ......................................................................... 84
      Metronome Garbage Collection policy .............................................................. 91
      How to do heap sizing ......................................... 92
      Interaction of the Garbage Collector with applications ......................................... 94
      How to coexist with the Garbage Collector ............................................................... 95
      Frequently asked questions about the Garbage Collector ..................................... 98
   Class loading .......................................................... 101
      The parent-delegation model ................................ 102
      Namespaces and the runtime package ................................................................. 103
      Custom class loaders ........................................... 103
   Class data sharing .................................................. 104
   The JIT compiler ..................................................... 105
      JIT compiler overview .......................................... 105
      How the JIT compiler optimizes code ................................................................. 106
      Frequently asked questions about the JIT compiler ........................................ 108
   The AOT compiler ................................................... 109
   Java Remote Method Invocation ................................ 110
      The RMI implementation ....................................... 110
      Thread pooling for RMI connection handlers ................................................... 111
      Understanding distributed garbage collection .................................................. 111
      Debugging applications involving RMI ............................................................... 112
   The ORB ............................................................. 112
   CORBA ............................................................... 113
   RMI and RMI-IIOP .................................................. 114
   Java IDL or RMI-IIOP? ........................................... 114
   RMI-IIOP limitations ............................................. 114
   Examples of client–server applications ....................... 115
   Using the ORB ...................................................... 119
   How the ORB works ............................................... 121
   Additional features of the ORB ................................ 129
   The Java Native Interface (JNI) ................................. 132
      Overview of JNI .................................................. 132
      The JNI and the Garbage Collector ............................................................... 133
      Copying and pinning ............................................. 138
      Handling exceptions ............................................ 139
      Synchronization .................................................... 140
      Debugging the JNI ............................................... 140
      JNI checklist ....................................................... 142

**Chapter 3. Planning** ................................................ 143
   Migrating from earlier releases of the IBM SDK, Java Technology Edition ..................... 143
   Version compatibility ............................................. 145
   Supported environments ........................................ 146
   Additional information for AIX ................................ 146

**Chapter 4. Installing and configuring the SDK and Runtime Environment** ............ 149
   Installing from an InstallAnywhere package .................... 150
      Completing an attended installation ................................................................. 150
      Completing an unattended installation ............................................................ 151
      Interrupted installation ........................................... 152
      Known issues and limitations ............................................... 153
      Relocating an instalp package ........................................... 153
      Upgrading the SDK ............................................... 153
      Verification ........................................................ 154
      Setting the path ...................................................... 155
      Setting the class path ............................................. 155
      Updating your SDK or runtime environment for Daylight Saving Time changes ........ 156

**Chapter 5. Developing Java applications** ....................... 157
   Using XML .......................................................... 157
      Migrating to the XL-TXE-J ......................................... 159
      Securing Java API for XML processing (JAXP) ................................................ 161
      against malformed input ........................................... 161
      XML reference information ......................................... 162
      Debugging Java applications ........................................ 166
      Java Debugger (JDB) ............................................ 167
      Determining whether your application is running on a 32-bit or 64-bit JVM ................ 168
   Porting Java applications to 64-bit systems ....................... 168
      General porting considerations ....................................... 168
      Porting native code from 32-bit to 64-bit systems ........................................ 171
      JNI considerations ................................................ 172
      Usage of the JNI interface ......................................... 174

© Copyright IBM Corp. 2011, 2018
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Garbage Collector diagnostic data</td>
<td>436</td>
</tr>
<tr>
<td>Class-loader diagnostic data</td>
<td>460</td>
</tr>
<tr>
<td>Shared classes diagnostic data</td>
<td>463</td>
</tr>
<tr>
<td>Using the HPROF Profiler</td>
<td>491</td>
</tr>
<tr>
<td>Using the JVMTI</td>
<td>496</td>
</tr>
<tr>
<td>Using the Diagnostic Tool Framework for Java</td>
<td>513</td>
</tr>
<tr>
<td>Using JConsole</td>
<td>520</td>
</tr>
<tr>
<td>Chapter 11. Reference.</td>
<td>525</td>
</tr>
<tr>
<td>Command-line options</td>
<td>525</td>
</tr>
<tr>
<td>Specifying command-line options</td>
<td>525</td>
</tr>
<tr>
<td>General command-line options</td>
<td>526</td>
</tr>
<tr>
<td>System property command-line options</td>
<td>527</td>
</tr>
<tr>
<td>JVM command-line options</td>
<td>549</td>
</tr>
<tr>
<td>JVM -XX: command-line options</td>
<td>569</td>
</tr>
<tr>
<td>JIT and AOT command-line options</td>
<td>573</td>
</tr>
<tr>
<td>Garbage Collector command-line options</td>
<td>577</td>
</tr>
<tr>
<td>Balanced Garbage Collection policy options</td>
<td>593</td>
</tr>
<tr>
<td>JVM messages</td>
<td>595</td>
</tr>
<tr>
<td>Finding logged messages</td>
<td>595</td>
</tr>
<tr>
<td>Obtaining detailed message descriptions</td>
<td>596</td>
</tr>
<tr>
<td>CORBA minor codes</td>
<td>596</td>
</tr>
<tr>
<td>org.omg.CORBA.BAD_OPERATION</td>
<td>596</td>
</tr>
<tr>
<td>org.omg.CORBA.BAD_PARAM</td>
<td>597</td>
</tr>
<tr>
<td>org.omg.CORBA.COMM_FAILURE</td>
<td>597</td>
</tr>
<tr>
<td>org.omg.CORBA.INTERNAL</td>
<td>598</td>
</tr>
<tr>
<td>org.omg.CORBA.MARSHAL</td>
<td>598</td>
</tr>
<tr>
<td>org.omg.CORBA.NO_RESPONSE</td>
<td>598</td>
</tr>
<tr>
<td>org.omg.CORBA.OBJ_ADAPTER</td>
<td>598</td>
</tr>
<tr>
<td>org.omg.CORBA.OBJECT_NOT_EXIST</td>
<td>598</td>
</tr>
<tr>
<td>org.omg.CORBA.TRANSIENT</td>
<td>599</td>
</tr>
<tr>
<td>org.omg.CORBA.UNKNOWN</td>
<td>599</td>
</tr>
<tr>
<td>Environment variables</td>
<td>599</td>
</tr>
<tr>
<td>Displaying the current environment</td>
<td>600</td>
</tr>
<tr>
<td>Setting an environment variable</td>
<td>600</td>
</tr>
<tr>
<td>Separating values in a list</td>
<td>600</td>
</tr>
<tr>
<td>JVM environment settings</td>
<td>600</td>
</tr>
<tr>
<td>Default settings for the JVM</td>
<td>603</td>
</tr>
<tr>
<td>Known issues and limitations</td>
<td>606</td>
</tr>
<tr>
<td>Support for virtualization software</td>
<td>609</td>
</tr>
<tr>
<td>Default Swing key bindings</td>
<td>609</td>
</tr>
<tr>
<td>Notices</td>
<td>623</td>
</tr>
<tr>
<td>Trademarks</td>
<td>625</td>
</tr>
<tr>
<td>Terms and conditions for product documentation</td>
<td>625</td>
</tr>
<tr>
<td>IBM Online Privacy Statement</td>
<td>626</td>
</tr>
</tbody>
</table>

Contents V
Preface

This guide provides general information about the IBM® SDK, Java™ Technology Edition, Version 7 Release 1, for all supported AIX® architectures. The guide gives specific information about any differences in the IBM implementation compared with the Oracle implementation.

Read this information in conjunction with the documentation on the Oracle website: [http://www.oracle.com/technetwork/java/index.html](http://www.oracle.com/technetwork/java/index.html).

Useful websites include:

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

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

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


Chapter 1. Product overview

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

This edition of the user guide applies to IBM SDK, Java Technology Edition, Version 7 Release 1, for all supported AIX architectures and to all subsequent releases, modifications, and service refreshes until otherwise indicated in new editions.

This guide applies to the following programs:
- IBM 64-bit SDK for AIX, Java Technology Edition, Version 7 Release 1

In this guide, the name IBM SDK, Java Technology Edition, Version 7 Release 1 is used to refer to any of the supported configurations, unless specifically called out in the text.

The IBM SDK, Java Technology Edition, Version 7 Release 1, for all supported AIX architectures provides new function when compared to IBM SDK, Java Technology Edition Version 7, while remaining fully compatible with Java SE Version 7. To learn about the new features and functions available, see “What’s new” on page 6.

This user guide is updated for each refresh of IBM SDK, Java Technology Edition, Version 7 Release 1 that contains an Oracle Critical Patch Update (CPU). Oracle CPUs are scheduled every 3 months, as outlined in the following bulletin: [https://www.oracle.com/technetwork/topics/security/alerts-086861.html](https://www.oracle.com/technetwork/topics/security/alerts-086861.html) IBM packages that contain the CPU are made available shortly afterward.

Late breaking information about the IBM SDK, Java Technology Edition, Version 7 Release 1 that is not available in the user guide can be found here: [https://www.ibm.com/support/docview.wss?uid=swg21639279](https://www.ibm.com/support/docview.wss?uid=swg21639279)

To determine the service refresh or fix pack level of an installed version, see “Obtaining version information” on page 207.

Any new modifications that are made to this user guide are indicated by vertical bars at the start of the changed lines.

Introduction

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 runtime 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. 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.

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

- The Java Compiler.
- The Java Runtime Environment and IBM J9 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 documentation in IBM Knowledge Center, 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/java71[_64]/bin directory.

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

**Contents of the SDK**

**SDK tools:**

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

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

- **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.

- **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 205.

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 253.

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

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

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

rmic (Java Remote Method Invocation (RMI) Stub Converter)
Generates stubs, skeletons, and ties for remote objects. Includes RMI over Internet Inter-ORB Protocol (RMI-IIOP) support.
rmid (RMI activation system daemon)
Starts the activation system daemon so that objects can be registered and activated in a Java Virtual Machine (JVM).

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

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

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

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

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

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

xjc
Compiles XML Schema files.

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 IBM API documentation for the SDK.

Copyright notice
The copyright notice for this release.

License file
The License file, /usr/swlag/<locale>/Java6_64.la, contains the license agreement for the SDK, 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.

Runtime Environment
The Java runtime environment provides runtime support for Java applications.

The runtime environment includes the IBM J9 Virtual Machine (JVM), which interprets Java bytecode at run time. There are a number of tools included with the runtime environment that are installed into the /usr/java71[_64]/jre/bin directory, unless otherwise specified.

Contents of the runtime environment
Core Classes
These classes are the compiled class files for the platform and must remain compressed for the compiler and interpreter to access them. Do not modify these classes; instead, create subclasses and override where you need to.
Trusted root certificates
From certificate signing authorities. These certificates are used to validate the identity of signed material.

Runtime environment tools

ControlPanel (Java Control Panel)
Configures your Runtime Environment.

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

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

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 205.

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 382.

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.

J9 Virtual Machine (JVM)
   The IBM J9 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 VM are:
   • JVM Application Programming Interface (API)
   • Diagnostic component
   • Memory management
   • Class loader
   • Interpreter
   • Platform port layer

   For further information about the JVM, see "Components of the IBM Virtual
Machine for Java” on page 67.

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:

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 Release 1</td>
<td>IBM J9 VM (build 2.7, JRE 1.7.0 ...</td>
</tr>
<tr>
<td>7</td>
<td>IBM J9 VM (build 2.6, JRE 1.7.0 ...</td>
</tr>
<tr>
<td>6</td>
<td>IBM J9 VM (build 2.4, JRE 1.6.0 IBM...</td>
</tr>
</tbody>
</table>

What's new

Learn about the new features and functions available with this release.

IBM SDK, Java Technology Edition, Version 7 Release 1 is a new release, that
extends the capabilities of IBM SDK, Java Technology Edition Version 7 service
refresh 6 and later refreshes. This section introduces new material for Version 7
Release 1.

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

For any updates to IBM security in this release, see Security Reference: What's new
First release

Learn about the new features and functions available with this release.

Version 7 Release 1 includes the following new features and important changes:

- Packed object technology preview
- “Data access acceleration API”
- “Enhancements to the JVM Dump API”
- “Improved JIT diagnostics”
- “Updates to Java core information”
- “New file format for Portable Heap Dumps (PHD)” on page 8
- “New JVMTI extensions for subscribing to tracepoints” on page 8
- “Improved performance of SDK method trace” on page 8
- “New tracepoint groups” on page 8
- “New IBM MXbeans for virtualized environments” on page 8
- “Enhanced verification errors” on page 8
- “Byte Code Instrumentation (BCI) implementation is changed” on page 8
- “Virtual memory released by default” on page 8
- “Change in JVM option string construction” on page 8
- “Removal of the Diagnostics Collector utility” on page 9
- “Deprecation of the -Xzero command-line option” on page 9
- “Default behavior change for method java.nio.ByteBuffer.allocateDirect(int)” on page 8
- “The Metronome Garbage Collector (GC) is now fully supported” on page 9

Packed object technology preview

Packed object support is available as a technology preview for evaluation purposes. This enhancement allows greater control over the layout of objects in memory. The capability enables greater flexibility when dealing with non-Java memory structures, for example, when exchanging and using data between Java and other languages or environments. For more information, see “Packed object evaluation technology” on page 16.

Data access acceleration API

A new API is available for manipulating native data. By using this API, you can simplify the processing of native data in an application, and take advantage of any hardware acceleration that is available on your system. For more information, see “The Data Access Accelerator Library” on page 197.

Enhancements to the JVM Dump API

The com.ibm.jvm.Dump API is enhanced so that you can specify dump file names when you trigger dumps at run time. You can also set and query JVM dump options by using the API. For more information, see the JVM Dump API.

Improved JIT diagnostics

A JIT dump is now produced following a general protection fault (GPF) or abort event. This binary dump helps IBM service diagnose any JIT problems that might be associated with application errors. For more information, see “JIT dumps” on page 328.

Updates to Java core information

The ENVINFO section of a Javadump now contains information about the
operating system hypervisor, if present. See “TITLE, GPINFO, and ENVINFO sections” on page 343 for examples.

New file format for Portable Heap Dumps (PHD)
The file format of portable heap dumps (PHD) is changed to PHD version 6. For more information, see “Portable Heap Dump (PHD) file format” on page 367.

New JVMTI extensions for subscribing to tracepoints
You can now subscribe to and unsubscribe from JVM tracepoints by using two new IBM JVMTI extensions. For more information, see “IBM JVMTI extensions - API reference” on page 499.

Improved performance of SDK method trace
Method trace performance is improved, which increases application throughput. The impact on application throughput is now proportional to the number of running methods that are being traced. For more information about this enhancement, see “Using method trace” on page 423.

New tracepoint groups
New tracepoint groups are provided for collecting existing diagnostic information, such as verbose garbage collection or -Xcheck output, as a trace stream. For more information, see “Tracepoint specification” on page 404.

New IBM MXbeans for virtualized environments
IBM MXBean extensions are available to provide information about virtualized environments that JVMs are running in. For more information about these MXBeans, see API documentation. For more information about using JConsole to view Mbeans and MXBeans, see “MBeans and MXBeans” on page 523.

Enhanced verification errors
The “-XX:[+|-]VerboseVerification” option routes data generated during verification from the class file StackMapTable attribute to stderr. This data provides extra contextual information, which can help with problem diagnosis.

Byte Code Instrumentation (BCI) implementation is changed
The -Xshareclasses:enableBCI suboption improves startup performance without using a modification context, when using JVMTI class modification. This option is now the default. If you are migrating from an earlier version of the IBM SDK, read the additional information that is provided in “Migrating from earlier releases of the IBM SDK, Java Technology Edition” on page 143.

Virtual memory released by default
When the object heap shrinks, the physical memory that is used for the object heap is now released to reduce the physical memory requirements of the process. On Linux, 4K pages are released. On z/OS, 4K and 1M pageable pages are released. Releasing large page sizes is not supported.

Note: There is no change in behavior for AIX or Windows; memory is already released by default.

Change in JVM option string construction
During startup, the JVM processes options in a different order from earlier releases. Options that are created automatically by the JVM, such as search
paths and version information, are constructed first and take the lowest precedence. For more information about construction order, see “Specifying command-line options” on page 525.

Removal of the Diagnostics Collector utility

The Diagnostics Collector utility is removed from this release onwards. Instead, use the IBM Support Assistant Data Collector to collect diagnostic data, and to send that data to IBM, if required. Use the -Xcheck:dump option to run the configuration checks that were previously provided by the Diagnostics Collector utility. For more information, see “IBM Support Assistant Data Collector” on page 436.

Deprecation of the -Xzero command-line option

The -Xzero option is deprecated in this release, and will be removed from future releases.

Default behavior change for method java.nio.ByteBuffer.allocateDirect(int)

The default behavior of the method java.nio.ByteBuffer.allocateDirect(int) changed in Java V7. Direct byte buffers are no longer allocated on a page boundary. Any applications that rely on the earlier undocumented alignment, which occurred in Java 6, can revert to the previous behavior with -XX:+PageAlignDirectMemory. For more information, see “JVM -XX: command-line options” on page 569.

The Metronome Garbage Collector (GC) is now fully supported

The Metronome GC, which was available as a technology preview in IBM SDK, Java Technology Edition, Version 7, is now fully supported in this release.

The Metronome 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. For more information about using this policy, see “Using the Metronome Garbage Collector” on page 218.

Service refresh 1

Skip to “Service refresh 1 fix pack 1” on page 10.

Service refresh 1

Learn about the new features and functions available with this release.

- “Default behavior change for unsigned or self-signed applets”
- “Increase in default size of ORB messages”
- “Thread trace history in Java dump files” on page 10
- “Improvements to native memory and thread information in system dumps” on page 10
- “Securing Java API for XML (JAXP) processing” on page 10
- “Enhanced trace messages for Java net native code” on page 10
- “CORBA debug enhancement” on page 10

Default behavior change for unsigned or self-signed applets

Unsigned or self-signed applets no longer run by default because the security settings are changed to high. For more information about this change, see “Default behaviour change when running unsigned or self-signed applets” on page 609.
Increase in default size of ORB messages
The default size of the com.ibm.CORBA.FragmentSize property is increased from 1024 to 4096 bytes. For more information about this property, see “Using the ORB” on page 119.

Thread trace history in Java dump files
If your Java dump file was triggered by an exception throw, catch, uncaught, or systhrow event, or by the com.ibm.jvm.Dump API, the dump file contains recent trace history for the current thread. For more information, see “Trace history for the current thread” on page 359.

Improvements to native memory and thread information in system dumps
Additional information is now available in system dumps to help with problem determination. You can use dump viewer commands to find information about native memory and information about running threads. For more information, see “Commands available in jdumpview” on page 384. You can also obtain this information by using the DTFJ API. For more information, see the DTFJ API reference.

Securing Java API for XML (JAXP) processing
You can control whether external entities are resolved in an XML document. This limit can be set on the command line by using a system property, or you can specify a value in your jaxp.properties file. You must also override the default XML parser configuration for the changes to take effect. For more information, see “Securing Java API for XML processing (JAXP) against malformed input” on page 161.

Enhanced trace messages for Java net native code
To improve problem diagnosis, new tracepoints are added for SocketException exceptions that occur during socket stream read and write operations. The trace component is net, and the tracepoint ID is net.501.

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

Service refresh 1 fix pack 1
This fix pack includes minor changes and fixes to the program code.

• “Change to Zambian currency code symbol”

Change to Zambian currency code symbol
In this release the currency code symbol for Zambia is corrected from “ZMK” to “ZMW”.

Service refresh 2
Skip to “Service refresh 2 fix pack 10” on page 12.

Service refresh 2
Learn about the new features and functions available with this release.

• “Support for hardware compression acceleration on Power Systems” on page 11
• “Specify a directory for all dump file types” on page 11
• “Improvements to Java dump output” on page 11
Support for hardware compression acceleration on Power Systems™

If your application uses java.util.zip to provide Java compression services, the Generic Work Queue Engine (GenWQE) accelerator for AIX can reduce CPU consumption and shorten processing times. Specific hardware and software prerequisites apply. For more information, see “Enabling hardware compression acceleration on Power Systems” on page 246.

Specify a directory for all dump file types

You can specify a directory to write all types of dump file to by using the new -Xdump:directory command-line option. This option enhances the existing ability to specify the dump file location for a particular dump agent type by using the -Xdump:<agent>:file option. You can use tokens in both options, as described in “Dump agent tokens” on page 335. For more information about the -Xdump option, see “Using the -Xdump option” on page 322.

Improvements to Java dump output

The THREADS section now contains more information about Java threads that are running, which helps with problem determination. For more information about these changes, see “Threads and stack trace (THREADS)” on page 351.

Changes to the access permissions for shared class caches

To enhance security, the virtual machine now runs access checks when a process attempts to access a shared class cache. These checks are in addition to the existing checks that are run by the operating system on file access (for persistent caches) and System V objects (for non-persistent caches). The access permissions for persistent shared class caches have also been modified, and are now the same as for non-persistent caches.

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

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

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

The following table summarizes the changes in access permissions for persistent shared class caches:
Table 1. Changes to the access permissions for persistent shared class caches

<table>
<thead>
<tr>
<th>Use of the -Xshareclasses:groupAccess command-line option at cache creation</th>
<th>Previous access permissions</th>
<th>New access permissions (now the same as non-persistent caches)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not specified</td>
<td>-rw--r---r--</td>
<td>-rw-----</td>
</tr>
<tr>
<td></td>
<td>Read/write for user, plus read-only for group and others</td>
<td>Read/write for user only</td>
</tr>
<tr>
<td>Specified</td>
<td>-rw-rw--r--</td>
<td>-rw-rw----</td>
</tr>
<tr>
<td></td>
<td>Read/write for user and group, plus read-only for others</td>
<td>Read/write for user and group only</td>
</tr>
</tbody>
</table>

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

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

- A user in a group creates a cache by using the -Xshareclasses:groupAccess option, then another user in the same group attempts to access the cache without using the -Xshareclasses:groupAccess option. In this situation, access is now denied. The second user must specify the -Xshareclasses:groupAccess option.

- A user attempts to access a persistent cache that was created by another user in a different user group, by using the -Xshareclasses:readonly option. Read-only access for group and other categories has been removed, so access is now denied. To enable access in this situation, create the cache by using the -Xshareclasses:cacheDir option, and set the permissions on the specified directory to allow read-only access to users who are outside the group of the cache creator.

For more information about shared classes command-line options, see “Class data sharing command-line options” on page 241.

Service refresh 2 fix pack 10

Learn about the new features and functions available with this release.

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

Reserving memory space for compressed references

A new option is available for securing space in memory for any native classes, monitors, and threads that are used by compressed references. Setting this option can help prevent OutOfMemoryError exceptions that might occur if the lowest 4 GB of address space becomes full. For more information, see “-Xmcrs” on page 587.

Ability to turn off the ALT-key function

A new system property is available that can prevent the ALT-key from highlighting the first menu in the active window. For more information, see “-Dibm.disableAltProcessor” on page 537.
Note: If your application uses a Windows Look and Feel (com.sun.java.swing.plaf.windows.WindowsLookAndFeel), this option has no effect.

Service refresh 3

Skip to “Service refresh 3 fix pack 10.”

Skip to “Service refresh 3 fix pack 20” on page 14.

Skip to “Service refresh 3 fix pack 30” on page 14.

Skip to “Service refresh 3 fix pack 40” on page 14.

Skip to “Service refresh 3 fix pack 50” on page 15.

Skip to “Service refresh 3 fix pack 60” on page 15.

Service refresh 3

This service refresh provides serviceability improvements. There is also a change in default behavior for applets and Web Start applications.

• “Improved tracing for the Object Request Broker (ORB)”
• “Change to applet or Web start application behavior”
• “Changes to InstallAnywhere unattended installation”

Improved tracing for the Object Request Broker (ORB)
Component level tracing is now available to improve the debugging of ORB problems. A new system property allows you to generate trace information for one or more ORB components, such as DISPATCH or MARSHAL. For more information about this system property, see “-Dcom.ibm.CORBA.Debug.Component” on page 529.

Change to applet or Web start application behavior
When an applet or Web Start application is started, a warning is displayed if a later release of the runtime environment is available that contains security updates. If the runtime environment is unable to determine whether a later release is available, a similar warning message is shown if the runtime environment is more than six months old.

Changes to InstallAnywhere unattended installation
If you use an existing response file to install packages without user intervention, you must include the line LICENSE_ACCEPTED=TRUE in the response file to indicate that you have read the terms of the license agreement and confirm your acceptance. This line is automatically added if you create a new response file by using the attended installation process. For more information, see “Installing from an InstallAnywhere package” on page 150.

Service refresh 3 fix pack 10

This service refresh includes serviceability improvements.

• “Invalidating AOT methods in the shared classes cache”

Invalidating AOT methods in the shared classes cache
You can now invalidate failing AOT methods in the shared classes cache to prevent them being loaded without destroying and re-creating the cache.
Three new `-Xshareclasses` suboptions are available to find, invalidate, or revalidate these methods. For more information, see “-Xshareclasses” on page 560.

**Service refresh 3 fix pack 20**

This service refresh includes additional operating system support and serviceability improvements.

- “Ability to check for the use of large pages”
- “AIX V7.2 support”
- “Changes to the `-Djava.security.debug` property”
- “New system property -Djdk.xml.max.xmlNameLimit”

**Ability to check for the use of large pages**

You can now check whether large pages are obtained for the object heap when they are requested by the `-Xlp:objectheap:pagesize` option. The `warn` suboption generates a warning message if large pages are not obtained and allows the process to continue. Alternatively, you can use the `strict` suboption to generate an error message and end the process if large pages are not obtained. For more information, see “-Xlp:objectheap” on page 556.

**AIX V7.2 support**

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

**Changes to the `-Djava.security.debug` property**

The `-Djava.security.debug` property is updated to include the missing access suboptions, which are `codebase` and `permission`. Three new suboptions are added to allow further customization of the output, namely `permname`, `permactions`, and `thread`. For more information about these suboptions, see the output from running `-Djava.security.debug=help`.

**New system property -Djdk.xml.max.xmlNameLimit**

If your application takes untrusted XML, XSD or XSL files as input, you can enforce specific limits during JAXP processing to protect your application from malformed data. The `-Djdk.xml.max.xmlNameLimit` option can be used to limit the length of XML names in XML documents. For more information, see “Securing Java API for XML processing (JAXP) against malformed input” on page 161.

**Service refresh 3 fix pack 30**

This fix pack contains only Oracle and IBM fixes to the code base.

**Service refresh 3 fix pack 40**

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

- “Changes to the shared classes cache generation number”

**Changes to the shared classes cache generation number**

The format of classes that are stored in the shared classes cache is changed due to an Oracle security update. As a result, the shared cache generation number is changed, which causes the JVM to create a new shared classes cache, rather than re-creating or reusing an existing cache. To save space, all existing shared caches can be removed unless they are in use by an

Service refresh 3 fix pack 50

Support for Java plug-in and Web Start features is removed in this release. This release contains Oracle and IBM fixes to the code base.

• “Support ends for Java plug-in and Web Start features”

Support ends for Java plug-in and Web Start features

As announced in the Oracle Java SE Support roadmap, support for Java plug-in and Web Start technology in Java SE 7 ends in July 2016. Accordingly, support is also removed from IBM SDK, Java Technology Edition, Version 7 Release 1. For continued support, move to the latest release of the IBM SDK for all applications launched or deployed through web browsers.

Service refresh 3 fix pack 60

This release contains the latest IBM fixes and the October 2016 Oracle Critical Patch Update (CPU). For more information about the Oracle CPU, see Critical Patch Updates, Security Alerts and Third Party Bulletin on the Oracle website.

Service refresh 4

Skip to “Service refresh 4 fix pack 5.”

Skip to “Service refresh 4 fix pack 10” on page 16.

Skip to “Service refresh 4 fix pack 15” on page 16.

Skip to “Service refresh 4 fix pack 20” on page 16.

Skip to “Service refresh 4 fix pack 25” on page 16.

Service refresh 4

This release contains the latest IBM fixes and the January 2017 Oracle Critical Patch Update (CPU).

Service refresh 4 fix pack 5

This release contains the latest IBM fixes and the April 2017 Oracle Critical Patch Update (CPU). There is also a change for .jar files signed with MD5.

.jar files signed with MD5 are treated as unsigned

To improve security, a new restriction is introduced in this refresh as part of the Oracle Critical Patch Update (CPU). Applications, including Applets or Web Start applications that use .jar files that are signed with MD5 are affected. These .jar files are treated as unsigned. To address the issue, the .jar file must be re-signed with a stronger algorithm or key size. For more information about this change, which includes a short term workaround, see the Oracle JRE and JDK Cryptographic roadmap.
Service refresh 4 fix pack 10
This release contains the latest IBM fixes and the July 2017 Oracle Critical Patch Update (CPU).

Service refresh 4 fix pack 15
This release contains the latest IBM fixes and the October 2017 Oracle Critical Patch Update (CPU).

Service refresh 4 fix pack 20
This release contains the latest IBM fixes and the January 2018 Oracle Critical Patch Update (CPU).

Security checking
To improve security, the SecurityManager is now enabled by default. You can disable security checking by setting the following system properties on the command line:

- `-Dcom.ibm.jvm.enableLegacyTraceSecurity=false`
- `-Dcom.ibm.jvm.enableLegacyDumpSecurity=false`
- `-Dcom.ibm.jvm.enableLegacyLogSecurity=false`

Service refresh 4 fix pack 25
This release contains the latest IBM fixes and the April 2018 Oracle Critical Patch Update (CPU).

Packed object evaluation technology
Packed object support is a new IBM enhancement for Java that is provided as a technology preview in this release. The content is available subject to a disclaimer.

Disclaimer
The code and application programming interfaces herein are technology preview information that may not be made generally available by IBM as or in a product. You are permitted to use the information only for internal use for evaluation purposes and not for use in a production environment. IBM provides the information without obligation of support and "as is" without warranty of any kind.

Packed objects
You can use packed objects to gain greater control over the layout of objects in memory. By using packed objects, your applications have greater flexibility when they work with memory structures that are not in Java code. For example, when interoperating between Java code and other languages or environments.

For general information about working with native memory structures, see "Advantages of using packed objects to access native data" on page 17.

You can also optimize some patterns of Java objects to reduce their overall size and improve the efficiency of access. For more information, see "Advantages of using packed objects to represent data" on page 18.
Advantages of using packed objects to access native data

Many Java applications and frameworks must deal directly with native data. Native data is memory that is allocated outside the Java heap. For example: operating system data structures, data that is transmitted, data that is streamed from a file, and data that is shared with another application that is not a Java application. By using packed objects, you can avoid the disadvantages of existing techniques for accessing native data, such as having to copy data between Java and native objects.

Existing methods of accessing native data in Java applications

You can access native data by using the following existing techniques:

- The Java Native Interface (JNI) API. You can use the JNI interface to copy all the information back and forth between native and Java data structures by using a native method. Alternatively you can leave the data entirely in native memory, and read and write to it by using native methods.
- The Java New I/O (NIO) API. You use this API to create java.nio.Buffer objects that interact with native memory.
- The sun.misc.Unsafe class. This class also provides functions to directly read and write to arbitrary memory locations (in native or Java code), but only if the caller has sufficient privileges.

By using these techniques, you avoid having to write lots of native methods, and the overhead of calling those methods. However, you must still marshall data into a Java object. For example, a direct ByteBuffer object can read a sequence of bytes into a byte array that is supplied by the caller, but this operation is a copy operation. Modifying the byte array afterward does not affect the native data that backs the ByteBuffer object, unless the data is explicitly copied back.

Many operating system data structures, file formats, and over-the-wire protocols consist of many fields of various types. To access these structures by using the methods described previously, you must know the offsets of each field and how to convert the binary representation into an equivalent Java format. For primitive data types, this conversion is usually trivial, but for more complex structures, you might have to create multiple objects to model the data.

A common technique is for a data structure to include fields that are also data structures, rather than simple primitive types. For example, a line in a two-dimensional space can be represented by the two endpoints. The endpoints can be represented as Point objects, which contain X and Y coordinates. For this example, a syntax that is similar to C might look as follows:

```c
struct Point {
    int x;
    int y;
}

struct Line {
    Point start;
    Point end;
}
```

The Point structures that are included in the Line object are referred to as being nested, or sometimes embedded or inlined. The space for the Point data is included in the containing structure, but you can access and operate on each Point structure separately if required.
To model this example in Java code, in an object-oriented manner, you require a Point class as well as a Line class. When you read a Line structure from native data, a total of 3 Java objects (one Line and two Point objects) are instantiated and initialized.

**Advantages of using packed objects to access native data**

By using packed objects, you can model native data structures, including nested structures, in Java code. This Java model overlays directly on the native memory, avoiding the costs of copying the data to and from Java code for each access operation.

**Advantages of using packed objects to represent data**

By using packed objects in your Java code, you can reduce the amount of memory that is required to store objects, and also increase the efficiency of caching.

In the Java memory model, all fields in an object are either a primitive data type, such as byte or int, or a reference or pointer to an object. Arrays of primitive data types, such as char[], are also objects. One disadvantage of this model is that, when you follow object-oriented design practices, data types are often composed of many different types in order to encapsulate both state and behavior. As a result, one data type can represent an entire tree of objects. The following costs are associated with having many objects:

- **Object overhead**
- **Object locality**

**Object overhead**

The first cost of having many objects is that each object in a JVM must have some metadata that is associated with it. For example, the java.lang.Class value that represents the type of that object, or the length of an array object. The most common approach is to place this metadata at the start of the object, creating an object header.

For a large or complex object, the size of the header is relatively insignificant. For a small object, however, the size of the header can become significant. Consider a byte array that contains a single element:

```java
new byte[1];
```

This object has only 8 bits of state, but the header must reference both the class (byte[]) and the length (1). The length must be represented by at least 32 bits because Java arrays are indexed by the int type. The class reference is also unlikely to be smaller than 32 bits. Therefore, 64 bits of metadata are required for a single 8-bit value. Additionally, the JVM is likely to add at least 3 bytes of padding to ensure that the subsequent object in the heap starts on an aligned address. The total extra memory requirement for 8 bits of data is therefore 88 bits.

Every object has a similar associated overhead, so the more objects you have, the greater the effect on system resources.

The structure of Java arrays can exaggerate this overhead. Consider an array of Point objects, representing coordinates in a two-dimensional space. Each instance of the Point class has two int values, $x$ and $y$, of 32 bits each, plus the object header. Assuming that the header is just the class reference, and occupies only 32 bits, each Point instance is 8 bytes of data and 4 bytes of extra overhead. An array
of 10 Point objects (new Point[10]) consists of the header (class + length = 8 bytes), plus 10 object references (assuming 4 bytes each = 40 bytes). If each element of the array contains a unique Point object, the total is 80 bytes of data, but 88 bytes of additional overhead.

**Object locality**

A second effect of having a data structure that is represented as a tree of objects arises from modern memory architecture. Modern hardware relies heavily on caching and prefetching to provide efficient access.

Caching exploits the observation that memory that was recently accessed is likely to be accessed again soon, so keeping the most recently accessed data in very fast memory usually results in the best performance. Data is cached in small blocks, which are known as cache lines, to exploit another observation: data that is stored in sequence is often accessed in sequence. Code that accesses array[i] often proceeds to access array[i+1].

When a data structure is composed of many different objects, an operation on the information might need to access several objects to locate the actual data. So for example, the implementation of Hashtable.get(obj) might contain code that looks as follows:

```java
int index = hash % length;
if(this.entries[index].key.equals(obj)) {
    return this.entries[index].value;
}
```

To complete this operation, the hashtable, the entries array, the chosen entry, and the key must all be loaded into memory. This memory loading consumes resources in the cache.

In addition, these objects cannot be guaranteed to be close enough in memory to appear in the same block of cached memory. Some JVM configurations attempt to keep related objects close to each other in memory, but this result is not always possible. Even when the JVM can place objects next to each other, the space that is required by the object header lies between the objects, possibly disrupting the benefit.

The chance of the memory that contains the fields entries[i].key and entries[i].value appearing in the same cache line are very high, but for entries[i].key and entries[i+1].key the chance is very low, even if the JVM can keep the objects close to each other.

**Optimizing Java code by using packed objects**

By using packed objects, you can exert more control over the layout of data structures in memory, without having to sacrifice an encapsulated design.

You can change fields that reference other data structures to include all the data for the field directly within the object. This change eliminates the overhead that is associated with the additional object, and ensures that the data is all stored contiguously in memory.

In the Point array example that was described previously, a packed array of Point objects does not require the 40 bytes of overhead that is incurred by the 10 Point
instances, or the 40 bytes of references to those instances. Only the overhead for a packed array is necessary, and this overhead is the same for an array of 10 Point instances or 10,000 Point instances.

In the hashtable example, if the array of entries is packed, the data for entries[i] and entries[i+1] is always adjacent in memory, and with no object header intervening. This structure can therefore increase the efficiency of caching.

**The packed object data model**

The packed object data model differs from the standard Java data model. Fields of primitive data type occupy the smallest amount of space necessary. Non-primitive data type fields can use packed types, in which the data is embedded rather than existing in a separate object. Elements of a packed array contain data rather than references to other objects. Packed objects can also contain data that is allocated outside of the Java heap.

**Primitive data type fields**

In the packed object data model, fields of primitive data type, such as byte or int, occupy the minimum amount of space necessary. Thus a field of type byte or boolean occupies a single byte; short or char occupies 2 bytes; int or float occupies 4 bytes; long or double occupies 8 bytes.

This model is in contrast to a standard Java object where all primitive data type fields are stored in either 32-bit (for byte, short, int, float, char, and boolean data types) or 64-bit format (for long and double data types). This model can be wasteful, especially when there are multiple fields of a small primitive data type.

For example:

```java
package com.ibm.packedexample.concepts;

@Packed
public final class RGBColour extends PackedObject
{
    public byte red;
    public byte green;
    public byte blue;
}
```

In the standard Java object model, the memory layout of this class is as follows:

```
0 4 8 12
Header red green blue
```

As a packed type, however, the object looks like this:

```
0 1 2 3 4
Header RG B
```

When this type is packed, it occupies only the 3 bytes that are required to represent the data. As with other objects, the JVM adds an object header, and a certain amount of padding to maintain object alignment, to any stand-alone instance of the RGBColour class.
Object type fields

Packed classes can also contain fields of non-primitive data types. For normal object and array types, these fields behave exactly like any other non-primitive data type field in the Java language; the field contains a reference (or a pointer), to either the field value or null. A packed object with one or more fields of this type is known as a mixed packed object, or mixed, and is subject to some additional restrictions.

Fields of packed types behave differently. An instance field of a packed type in a packed class does not result in a reference to another object, but instead the data fields of that type are embedded. A field of this type is known as a nested field, or nested.

For more information about nested fields, see Nested fields.

Arrays

In many circumstances, the benefit of data packing is minimal when you are working with a single instance of a class, but can be significant when you are working with large numbers of them. As with fields, the elements of an array of a packed type are packed into the array. This behavior is in contrast to a standard Java array, where the elements are references which point to the values.

For example, a packed array of 10 RGBColour objects uses only 30 bytes of data, plus the array object header and any padding that is added by the JVM. In the standard Java object model, the array would consist of 10 object references, probably of 4 bytes each, plus 10 instances of the RGBColour class, each consisting of 12 bytes of data plus an object header.

For more information about packed data arrays, see Packed data arrays.

Off-heap packed objects

The previous examples show objects that are allocated on the Java heap. You can also create a packed object in native memory, in other words, outside the Java heap. In these cases, the object that Java interacts with is very small, and consists primarily of a pointer to the actual data. These objects are known as off-heap packed objects. Aside from specific queries, the differences between an on-heap and off-heap packed object are not visible to the developer.

As an off-heap packed object, the RGBColour instance that was shown previously looks like this:

Developing with packed objects

The following topics include information about creating packed objects and using them in your code.

- "Packed class declaration" on page 22
Packed class declaration:

Use the @Packed annotation to indicate that a class should have its data packed. Several restrictions apply, to the class itself and to its fields and methods.

The following restrictions apply for declaring packed classes:

- The class must extend the PackedObject class or an abstract @Packed class.
- The class must be marked final or abstract.
- Instance methods cannot be marked synchronized. The class can, however, contain static synchronized methods.
- Instance fields cannot be marked transient or volatile.
- Nested instance fields cannot be marked final. For more information, see Nested fields.

Note: You can mark an inner class with the @Packed annotation, but caution is required. If the class is not marked static, the Java compiler inserts an additional, hidden field of type Object into the declaration, to hold the enclosing instance. This behavior changes the shape of the data in the packed object, and causes the object to become a mixed packed object.

When you use a packed class within another class, you must declare the class by using the @ImportPacked annotation. For more information, see “Use of packed types in other classes.”

Use of packed types in other classes:

If you use a packed type in a class, you must declare the packed type by using the @ImportPacked annotation.

By default, types are assumed to not be packed. To use a packed type in either a packed or a non-packed class, you must declare that packed type in advance. When the class is compiled, the compiler can then determine whether any packed types are used, without having to first load all the referenced type classes. This result improves performance.

To declare a packed type, specify the @ImportPacked annotation on the referencing class. In the annotation, list all the packed types that you will use in the class. You might use packed types as fields, parameters, or return types. If you intend to use an array of a packed type, you must specify that packed type. Use fully qualified class names, and the forward slash (/) as the delimiter character, instead of period (.). This format is the internal class name that is used by the JVM.
The @ImportPacked annotation applies to a single class; subclasses do not inherit imports from their parent class, and must specify their own @ImportPacked annotation. Similarly, inner and outer classes must also specify their own @ImportPacked annotation.

When declaring a @Packed class that extends an abstract @Packed class, the abstract @Packed superclass must be declared in the @ImportPacked annotation. For more information, see "Abstract packed classes" on page 34.

For example:

```java
@ImportPacked({
    "com/ibm/jvm/packed/types/PackedInt",
    "com/ibm/jvm/packed/types/PackedInt$Array",
    "com/ibm/jvm/packed/types/PackedLong",
    "com/ibm/jvm/packed/types/PackedLong$Array",
    "A$C"
})
class A {
    PackedInt.Array packedIntArray;
    PackedLong.Array packedLongArray;
    PackedObject[] packedObjArray;
    C packedC;
}

@ImportPacked({
    "com/ibm/jvm/packed/types/PackedInt",
    "com/ibm/jvm/packed/types/PackedInt$Array"
})
class B {
    PackedInt.Array packedIntArray2;
}

@Packed
static final class C extends PackedObject {
    int i;
    int j;
}
```

Notes: In this example:
- The @ImportPacked annotation is required on class A because class A uses packed types. Class A is not itself a packed class.
- The com/ibm/jvm/packed/PackedObject class is not listed in the @ImportPacked annotation because the PackedObject class is not a packed type. For more information, see "The PackedObject type hierarchy" on page 39.
- The inner class, class B, also requires an @ImportPacked annotation because it uses a packed type.
- The inner class, class C, does not require an @ImportPacked annotation because it does not use packed types.
- To use a packed array class, such as PackedInt.Array or PackedLong.Array, both the element class and the array class must be declared in the @ImportPacked annotation.

Nested fields:

When a packed class contains an instance field that is a packed type, the data for that field is packed directly into the containing class. The field is known as a nested field. When reading from a nested field, a small object is created as a pointer to the data. When writing to a nested field, you must use an explicit copy operation.
A nested packed array is a type of nested field. For more information, see “Nested packed arrays” on page 33.

Consider a class that represents a simple two dimensional box that you might use in a graphical user interface or drawing program. The class consists of two points: the start point and the dimensions of the box. The class also has two fields that represent the colors to use for the frame of the box, and the interior fill.

Here are some examples:

Point.java:
package com.ibm.packedexample.concepts;

@Packed
public final class Point extends PackedObject
{
    public int x;
    public int y;
}

Box.java:
package com.ibm.packedexample.concepts;

@ImportPacked({
    "com/ibm/packedexample/concepts/RGBColour",
    "com/ibm/packedexample/concepts/Point",
})
@Packed
public final class Box extends PackedObject
{
    public Point origin;
    public Point extent;
    public RGBColour frameColour;
    public RGBColour fillColour;
}

Note: RGBColour.java is described in The packed object data model.

If Box is a standard Java type, the class would consist of four object pointers, with all the data stored in those objects. The following diagram represents the memory layout:
As a packed type, however, the representation is far more compact. The only object is the Box, which is represented in the following diagram. These four fields are all of packed types, so their data is directly embedded into the object:

Note: Fields of primitive data types, such as int, are not considered nested because they are not object types. Similarly, fields of non-packed object types, such as String are also not nested. Instead of their data being embedded in the packed object, such fields are referenced from within the packed object, in the same way that they would be from a standard Java object.

You do not need to allocate and initialize nested fields. Rather than being initially null, all the data is available and initialized to the default value. See "Initialization and construction" on page 37.

Reading nested fields

When accessing a nested field in a packed object, the JVM must return an object for further operations to interact with. Because the data for the field is contained within the object, a small object must be generated to describe the data. This object is known as a derived object. This object is not a copy of the data, but a pointer to the data.

The following diagram illustrates accessing the box.extent derived object:
Derived objects tend to be short lived, and in many cases can be removed entirely by the just-in-time compiler (JIT). For example:

```java
box.extent.y
```

The derived object for the `box.extent` fields can be removed by compiler optimization because the appropriate offset into the object data can be calculated directly. In the example, this offset is offset 12 into the data.

**Writing to nested fields**

Because nested fields consist of data and not object pointers, you cannot assign an object into a nested field. The data can be copied in, but the reference to the object cannot be preserved. Likewise, further changes to the stored object are not reflected in the field, as these changes would be if storing a reference. Therefore, direct assignment into nested fields is not allowed. Instead, you must use an explicit copy operation with the `PackedObject.copyFrom(PackedObject)` method. This method copies the data from the argument into the receiver.

The following example is not valid when implementing a method like `setFillColour(RGBColour)`:

```java
void setFillColour(RGBColour value)
{
    fillColour = value; // Invalid
}
```

Use the following example instead:

```java
void setFillColour(RGBColour value)
{
    fillColour.copyFrom(value);
}
```

For more information about assignment to nested fields, see [Assignment Semantics](#).

An additional consequence of assignment is that `final` nested fields are not allowed. This rule exists because of a conflict between the compiler and the JVM:

- The compiler expects the field to be initialized, that is, assigned to, before the constructor completes.
- The JVM has implicitly initialized the field along with the containing object, and prevents assignment to the field.

The `final` modifier has no effect. Because nested fields cannot be assigned to, the fields are implicitly final. However, the contents of the nested field can be changed, which is standard Java behavior. Consider a final field of type `Object[]`. The reference to the array cannot be changed after it is assigned, but the contents of the array can be freely modified.

**Packed arrays**

Packed arrays represent collections of objects more compactly than standard Java arrays. Although a single instance of a packed class might be only slightly more compact than a single instance of the equivalent non-packed class, the benefits of data packing are amplified when you are working with arrays of instances.

The elements of a standard Java array are references, which each point to a separate object that contains the element data. In contrast, the elements of a packed array are contained directly inside the array, similar to the fields of a packed...
object. When an application reads from a packed array, a small object is created as a pointer to the data. For an application to write to a packed array, you must use an explicit copy operation.

The following example class, Complex, represents a complex number. The class is a packed class with two fields of type double:

```java	package com.ibm.packedexample.concepts;

import com.ibm.jvm.packed.Packed;
import com.ibm.jvm.packed.PackedObject;

@Packed
public final class Complex extends PackedObject {
  private double real;
  private double imaginary;
}
```

This packed class does not differ significantly from a standard Java class that is declared in the same way. However, a packed array of Complex objects differs significantly from a standard array of Complex objects.

A standard array of Complex objects consists of an array header and the pointers to the elements. These pointers are initially null, and must be initialized to point to instances of the Complex class. Each instance of the Complex class consists of a header and the two double fields, as shown in the following diagram:

```
Header 0: real 1: real ... n: real
0 8 16

Header 0: imaginary 1: imaginary ...
0 8 16
```

A packed array of Complex objects is far more compact. An array header is still present, but rather than pointers to the elements, the data for each element is part of the array, as shown in the following diagram:

```
Header 0: real 0: imaginary 1: real 1: imaginary ...
0 8 16
```

You do not need to initialize the elements of a packed array. Rather than being null, all the data is available and initialized to the default value. See “Initialization and construction” on page 37.
Packed array classes

Packed array classes are defined differently from standard array classes. You must define each packed array class explicitly in Java code. You must define a packed array class as a static packed nested class of its element class. For more information, and examples, see “Packed array classes” on page 31.

The com.ibm.jvm.packed.reflect.PackedArray class implements methods for working with packed arrays. For example:

- Allocating packed arrays
- Accessing array elements
- Obtaining the length of a packed array

The following code shows the definition of the packed array class for the Complex class:

```java
package com.ibm.packedexample.concepts;
import com.ibm.jvm.packed.ImportPacked;
import com.ibm.jvm.packed.Packed;
import com.ibm.jvm.packed.PackedObject;
import com.ibm.jvm.packed.reflect.PackedArray;

@Packed
public final class Complex extends PackedObject {
    private double real;
    private double imaginary;

    // Packed array class definition
    @ImportPacked({"com/ibm/packedexample/concepts/Complex"})
    @Packed
    public static final class Array extends PackedObject implements Serializable, Cloneable {
        private Array() {
        }
        public static Array allocate(int length) {
            return PackedArray.newArray(Array.class, length);
        }
        public Complex at(int index) {
            return PackedArray.at(this, index);
        }
        public int getLength() {
            return PackedArray.getLength(this);
        }
        public Array clone() throws CloneNotSupportedException {
            return (Array)super.clone();
        }
    }
}
```

Using packed arrays

The following example class, ComplexVector, represents a vector of complex numbers. This example uses a packed array of instances of the Complex class that was defined in the first example:

```java
package com.ibm.packedexample.concepts;
@ImportPacked({
```
"com/ibm/packedexample/concepts/Complex",
"com/ibm/packedexample/concepts/Complex$Array"
})
public class ComplexVector {
    protected Complex.Array elements;

    public ComplexVector(int length) {
        elements = Complex.Array.allocate(length);
    }
}

Although this ComplexVector class is not a packed class, it uses two packed classes: Complex.Array and Complex. The ComplexVector class must therefore have an @ImportPacked annotation to declare these packed classes. For more information, see "Packed array classes" on page 31 and "Use of packed types in other classes" on page 22.

Allocating packed arrays

To allocate a packed array, use the static allocate() method of the packed array class, or use the factory methods in the com.ibm.jvm.packed.reflect.PackedArray class.

The following example allocates a packed array of Complex objects by using the static allocate() method:
Complex.Array elements = Complex.Array.allocate(10); // allocate a packed array with 10 elements

The following example allocates an on-heap packed array of Complex objects by using a factory method:
Complex.Array elements = PackedArray.newArray(Complex.Array.class, 10);

The following example allocates an off-heap packed array of Complex objects by using a factory method:
Complex.Array elements = PackedArray.newNativeArray(Complex.Array.class, 10);

Note: You cannot allocate packed arrays by using the new[] operator.

Reading from packed arrays

To access an element of a packed array, use the at() method of the packed array class. Because packed array classes are not standard array classes, you cannot use the [] operator.

The following code is an example of reading the first element of a packed array of Complex objects:
Complex c = elements.at(0);

When an application accesses an element from a packed array, the JVM must return an object for further operations to interact with. Because the data for the element is contained within the array, a small object must be generated to describe the data. This object is known as a derived object. This object is not a copy of the data, but a pointer to the data.

Accessing elements.at(1) is illustrated in the following diagram:
 Derived objects tend to be short lived, and in many cases can be removed entirely by the just-in-time (JIT) compiler. For example:

```java
elements.at(i).imaginary
```

The derived object for `elements.at(i)` can be removed by compiler optimization because the appropriate offset into the array data can be calculated directly.

**Writing to packed arrays**

Because packed array classes are not standard array classes, you cannot initialize packed arrays by using initializer syntax, such as in the following code:

```java
String[] args = {"a", "b"};
```

Because packed arrays contain data and not object pointers, you cannot assign an object into a packed array. The data can be copied into the array, but the reference to the object cannot be preserved. Likewise, further changes to the stored object are not reflected in the array, as these changes would be if a reference was stored. Therefore, direct assignment into a packed array is not allowed. Instead, you must do an explicit copy operation by using the `PackedObject.copyFrom(PackedObject)` method. This method copies the data from the argument into the receiver.

The following example is not valid when you are implementing a method such as `setElement(int, Complex):

```java
elements.at(index) = value;  // Invalid code
```

Use the following example code instead:

```java
elements.at(index).copyFrom(value);
```

For more information about assignment to packed array elements, see [Assignment Semantics](#).

**Multi-dimensional arrays**

Multi-dimensional arrays of packed types are not allowed because a packed array might represent a view of data that is not in the Java heap. You might not be able to address an array that is allocated in one language or system in the same way as you can with Java. For example, a two-dimensional array of `int` values might be stored as rows then columns, or columns then rows.

**Standard arrays of packed arrays**

The following example shows a standard array of packed arrays. Each element of the `array1dOfPackedArrays` object is a reference to a packed array of type `Complex.Array`:

```java
Complex.Array[] array1dOfPackedArrays = new Complex.Array[10];
for (int i = 0; i < array1dOfPackedArrays.length; i++) {
    array1dOfPackedArrays[i] = Complex.Array.allocate(i);
}
```
The following example shows a two-dimensional standard array of packed arrays. Each element of the array2dOfPackedArrays object is a reference to a standard array of type Complex.Array[]:`

```java
Complex.Array[][] array2dOfPackedArrays = new Complex.Array[3][];
for (int i = 0; i < array2dOfPackedArrays.length; i++) {
    array2dOfPackedArrays[i] = new Complex.Array[4];
    for (int j = 0; j < array2dOfPackedArrays[i].length; j++) {
        array2dOfPackedArrays[i][j] = Complex.Array.allocate(j);
    }
}
```

**Packed array classes:**

Unlike standard Java technology array classes, if you want to collect packed objects into a packed array, you must define a packed array class inside the corresponding packed object class. You can define packed array classes only for final packed classes.

**Packed array class definition**

The following example illustrates the definition of a packed array class that corresponds to a packed class named com.ibm.packedexample.concepts.ExamplePacked. The code in italic font is specific to this packed class. Use the remainder of the code for all packed array classes.

```java
// Example package:
package com.ibm.packedexample.concepts;

import java.io.Serializable;
import com.ibm.jvm.packed.reflect.PackedArray;
import com.ibm.jvm.packed.ImportPacked;
import com.ibm.jvm.packed.PackedObject;

@Packed
// Example outer class:
public final class ExamplePacked extends PackedObject {
    // For a different packed array class, replace
    //com/ibm/packedexample/ExamplePacked with the
    // fully qualified name of the outer class
    @ImportPacked({"com/ibm/packedexample/ExamplePacked"})
    @Packed
    // The nested packed array class definition:
    public static final class Array extends PackedObject implements Serializable, Cloneable {
        private Array() {
        }
        public static Array allocate(int length) {
            return PackedArray.newArray(Array.class, length);
        }
        // For a different packed array class, replace ExamplePacked with
        // the type of the outer class.
        public ExamplePacked at(int index) {
            return PackedArray.at(this, index);
        }
        public int getLength() {
            return PackedArray.getLength(this);
        }
    }
}
```
public Array clone() throws CloneNotSupportedException {
    return (Array)super.clone();
}

// You can define other members of the @Packed outer class as usual
int field; // a field of the ExamplePacked class
...

You must implement a packed array class as a static, nested class. The outer class is the class of the elements in the packed array. The outer class must comply with the following conditions:
- The class must be declared a packed class, by using the @Packed annotation.
- The class must be a subclass (direct or indirect) of the PackedObject class. For example, the class can be a subclass of a subclass of the PackedObject class.
- The class must be declared final.

The nested class must comply with the following conditions:
- The class must be named Array.
- The class must be declared final, static, and public.
- The class must declare the outer class by using the @ImportPacked annotation.
- The class must be declared a packed class, by using the @Packed annotation.
- The class must directly extend the PackedObject class.
- The class must have exactly one constructor, which must be private and take no arguments.
- The class must implement the Serializable and Cloneable interfaces.

Additionally, the nested class must implement the allocate(), at(), getLength(), and clone() methods exactly as shown in the previous example, except for the example text in italic font. The nested class cannot contain any other methods.

The nested class cannot contain any fields.

The com.ibm.jvm.packed.reflect.PackedArray class is supplied in this release. The class implements operations on packed arrays.

Using a packed array class

To use a packed array class, you must declare both the element (outer) class and the packed array class in an @ImportPacked annotation.

The following code uses a packed array of ExamplePacked instances:

```java
import com.ibm.packedexample.concepts.ExamplePacked;
import com.ibm.jvm.packed.ImportPacked;

// Declare the element class and the packed array class
@ImportPacked({
    "com/ibm/packedexample/concepts/ExamplePacked",
    "com/ibm/packedexample/concepts/ExamplePacked$Array" })
class TestPackedArray {
    void testArray() {
        // Allocate a packed array of 10 elements
        ExamplePacked.Array earray = ExamplePacked.Array.allocate(10);

        // Set the field of each packed array element to 0
        for (int i = 0; i < earray.getLength(); i++) {
```
Nested packed arrays:

When you nest a packed array into a packed object, the packed array must be of a constant size, which you specify by using the @Length annotation. If the array is of primitive data types, you must use packed classes that are equivalent to the primitive data types.

The "Nested fields" on page 23 topic introduces how to nest packed objects. The "Packed arrays" on page 26 topic introduces standalone packed data arrays. You can also nest a packed array into a packed object.

The size of a stand-alone packed array is arbitrary, and can be specified when the packed array is allocated. In contrast, the size of a nested packed array must be constant, and must be specified in the definition of its containing class. This restriction is necessary for the following reasons: for non-array objects in the Java language, any two instances of the same class must have the same amount of data, and the fields within that class must be at the same offsets. If nested packed arrays of different sizes were allowed, an array in the middle of an object would make the size of that object, and the offsets of the fields in the object, unpredictable. C and C++ programs have a similar restriction in struct and class declarations.

You specify the size of a nested packed array by using the @Length annotation.

In the following example, the ColourPalette class represents a simple 16-color indexed palette, where each color entry is represented by an RGBColour object.

ColourPalette.java:

```java
package com.ibm.packedexample.concepts;

@ImportPacked({
   "com/ibm/packedexample/concepts/RGBColour"
   "com/ibm/packedexample/concepts/RGBColour$Array"
})
@Packed
public final class ColourPalette extends PackedObject {
   @Length(16) public RGBColour.Array palette;
}
```

The @Length(16) annotation declares that the array is a nested packed array of 16 elements. The @Length annotation is valid only on fields that are arrays of a packed type. You get a verification error if you attempt to apply a @Length annotation to a non-packed or non-array field.

This restriction means that you cannot annotate arrays of primitive data types with the @Length annotation. The following example is not allowed:

```java
@Length(32) public byte[] bytes;
```

If you use a primitive data type array, without the @Length annotation, as a field in a packed object, the array is not nested. The array is a standard Java reference.
The packed primitive data type classes

Because you cannot annotate arrays of primitive data types with the @Length annotation, you must use packed classes that are equivalents of the primitive data types. Each class declares a single field, named value, of the appropriate type.

<table>
<thead>
<tr>
<th>Primitive data type</th>
<th>Packed equivalent class</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>PackedBoolean</td>
</tr>
<tr>
<td>byte</td>
<td>PackedByte</td>
</tr>
<tr>
<td>char</td>
<td>PackedChar</td>
</tr>
<tr>
<td>double</td>
<td>PackedDouble</td>
</tr>
<tr>
<td>float</td>
<td>PackedFloat</td>
</tr>
<tr>
<td>int</td>
<td>PackedInt</td>
</tr>
<tr>
<td>long</td>
<td>PackedLong</td>
</tr>
<tr>
<td>short</td>
<td>PackedShort</td>
</tr>
</tbody>
</table>

These equivalent types are also useful for accessing blocks of native data, typically by using the PackedObject.newNativePackedArray() API.

The following example, which uses the packed primitive data type class PackedByte, creates the nested array correctly:

```
@Length(32) public PackedByte.Array bytes;
```

Abstract packed classes:

You can declare a packed class as abstract. You can also extend an abstract packed class to define other packed classes. When you extend an abstract packed class, declare the parent class in the subclass by using the @ImportPacked annotation.

In an instance of a packed class, the instance fields are laid out in the order that they are declared. Instance fields that are declared in the superclass are first, followed by instance fields that are declared in the class itself. Padding might be automatically inserted to align reference fields. For example:

```
@Packed
abstract class A extends PackedObject {
   int a1;
   int a2;
}

@ImportPacked({"A"}) // declare B's superclass as @Packed
@Packed
final class B extends A {
   int b;
}
```

The layout of an instance of B class is as follows:

```
0  4  8 12
Header a1 a2 b
```

You cannot instantiate an abstract packed class.
You cannot use an abstract packed class as the type of a nested field because a nested field is implicitly instantiated when its containing class is instantiated.

You cannot instantiate a packed array of an abstract packed class because the elements of a packed array are implicitly instantiated when the packed array is instantiated. However, you can instantiate a standard array of an abstract packed class.

Examples

The following packed class definition of InvalidConcreteClass is invalid because it contains a nested field whose type is an abstract packed class:

```
@Packed
abstract class AbstractClass extends PackedObject {
    int i;
}

@ImportPacked({"AbstractClass"})
@Packed
final class InvalidConcreteClass extends PackedObject {
    AbstractClass nested_abstract;  // illegal nested field
}

@ImportPacked({"AbstractClass"})
@Packed
final class ValidConcreteClass extends AbstractClass {
}
```

The following code fails because an abstract packed class cannot be the element type of a packed array class:

```
PackedObject packedArray = PackedArray.newArray(PackedArray.getPackedArrayClass(AbstractClass.class), 10);  // error
```

The following code is valid, and creates a standard array of references:

```
AbstractClass[] refArray = new AbstractClass[10];
refArray[0] = new ValidConcreteClass();
```

Assignment semantics:

When you operate on packed objects, you must follow specific assignment rules. These rules have implications for constructing and initializing objects.

In the Java programming model, all values are either object type or primitive data type:

- **Object type**
  - Object values are always accessed by reference, and assignment to a field or array element of object type is by reference. Examples of object types include instances of the java.lang.Object class, or a subclass of that class.

- **Primitive data type**
  - Primitive data types are always accessed by value, and assignment to a field or array element of primitive data type is by value. Examples of primitive data types include byte or int.

An example of assignment by value:
/ By value semantics
int[] array = new int[1];
int value = 100;
array[0] = value;
value = 150;
return array[0];

The last line returns the value 100. The assignment to value has no effect on data in the array.

An example of assignment by reference:
// By reference semantics
Point[] array = new Point[1];
Point point = new Point(100, 200);
array[0] = point;
point.setX(150);
return array[0].getX();

The last line returns the value 150. The array element and the point object both reference the same value.

Elements of packed arrays and nested fields are not separate objects. Access to these arrays or fields might cause the creation of temporary object headers, but the actual data remains in the containing object. Therefore, the result of assigning a value to a nested field or a packed array element might not be obvious. For example, in the third line of the following code, is the assignment by-reference or by-value? What does the last line return?
PackedInt.Array array = PackedInt.Array.allocate(1);
PackedInt packed = new PackedInt(100);
array.at(0) = packed; // What does this mean?
packed.value = 150;
return array.at(0).value;

In this example, if the assignment is done by value, the return value is 100 because changing the value of the packed instance to 150 does not affect the copy of the packed instance that is stored in the array. A packed array has space for all contained data, so the reference cannot be stored in the traditional sense.

To avoid confusion, direct assignment to nested fields and packed array elements is not allowed. Instead, you must use the copyFrom() API, which provides a clear indication that assignment-by-value semantics are being used. The following example shows how you might use the copyFrom() method in your code:
PackedInt.Array array = PackedInt.Array.allocate(1);
PackedInt packed = new PackedInt(100);
array.at(0).copyFrom(packed);
packed.value = 150;
return array.at(0).value;

In this example, the use of the copyFrom() method in the third line is a clear indication that the assignment is by-value. The return result is still 100. Only the array assignment is changed. The assigning variables, such as array and packed, remain unchanged, as do stores to non-nested fields, such as value. Only assignments to nested fields and packed array elements are affected.

The following example also shows the nested field case:
@Packed
class A
{
    int i;
```java
int j;
}

@ImportPacked(\{"A\})
@Packed class B
{
    public float f;
    public A a;
    public String s;
}

B b = new B();
b.f = 3.14159; // Direct assignment
b.s = "pi"; // Direct assignment
b.a.i = 10; // Direct assignment to field A.i
b.a.j = 20; // Direct assignment to field A.j

A a = new A();
a.i = 123; // Direct assignment
a.j = 321; // Direct assignment

// b.a = a; // Not valid
b.copyFrom(a); // Correct
```

Related information:

"Packed objects glossary" on page 44
Glossary for the packed object support.

Initialization and construction:

Because of the way that packed objects are assigned, there are implications for object initialization and construction. You should separate initialization code into an instance method.

Nested fields in a packed object are not references, therefore the nested fields cannot be initialized to null like a regular field of object type. Instead, all the fields of the nested object are initialized to the appropriate initial value, which might be 0, false, or null. No constructor method is called on the nested objects.

The factory methods, such as PackedObject.newPackedObject(), also leave all fields in the default state without calling a constructor method. When constructing a native packed object with a non-zero pointer, that is, on existing memory, no initialization is done. This behavior is critical because the purpose is to interact with a data type that already exists outside the Java heap.

To follow best practices, you should separate initialization code into an instance method. This initialize method can be called from the constructor method, but also allows a containing object to call an appropriate initialize method on a nested object.

To continue with the earlier example that uses class A, you might implement the following lines of code:

```java
public A(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

A constructor method for class B might include the following lines of code:
public B(int x, int y)
{
    f = (float)x / (float)y;
    a = new A(x, y);
}

However, in this case, the assignment to object a is not allowed. Here is a possible workaround, although you must create a new object just to initialize the field:

```java
public B(int x, int y)
{
    f = (float)x / (float)y;
    a.copyFrom(new A(x, y));
}
```

If you add an initializer method to class A, you can use that method in the constructor method for class B. For example:

```java
public A(int x, int y)
{
    initialize(x, y);
}
public void initialize(int x, int y)
{
    this.x = x;
    this.y = y;
}
public B(int x, int y)
{
    f = (float)x / (float)y;
    a.initialize(x, y);
}
```

**Limitations of the packed model:**

Because multiple objects might refer to the same underlying data, limitations apply when you use equality and hash codes, synchronization, and finalization.

**Equality and hash codes**

The issue of identity, equality, and hash codes is related to the assignment issue described in "Assignment semantics" on page 35. Access to nested fields and elements of packed arrays might involve object headers being created dynamically, which can lead to some differences in behavior. The following example uses the classes from the "Assignment semantics" on page 35 topic:

```java
B b = new B();
A a1 = b.a; // fetch the nested field
A a2 = b.a; // fetch the nested field again
return a1 == a2; // returns false
```

The equality comparison (==) is not meaningful for packed objects, and should be avoided.

To determine whether two packed objects refer to the same underlying data, use the isIdentical() method on PackedObject.

The default implementation of the equals() method on PackedObject is the same as isIdentical(). You can optionally override equals() to provide your own semantics.

The default implementation of the hashCode() method on PackedObject, and the implementation of System.identityHashCode() for packed objects, is based on
identity as defined by isIdentical(). Again, you can optionally override hashCode() to provide your own semantics.

**Synchronization**

When you use packed objects, multiple objects might refer to the same underlying data. This data might be outside the Java heap. Trying to resolve contention between multiple objects, or between Java code and other types of code, is outside the scope of the Java synchronization model. Therefore, synchronization on packed objects is not supported. Declaring a synchronized instance method on a packed class, or using a synchronized block on a packed object, generates an error.

The methods wait(), notify(), and notifyAll(), which are commonly inherited from the Object class, are not available on the PackedObject class.

A potential strategy is to use separate lock objects to control access to all objects that share underlying data. For on-heap packed objects, separate non-packed objects might be used as locks. For off-heap data, a locking mechanism that can be shared between the Java and native code might be required, such as an operating system lock.

Synchronized static methods are allowed on packed classes because these methods operate on the java.lang.Class object, which is not packed.

**Finalization**

A common use of finalization is to handle off-heap memory management, for example, freeing a native memory buffer when the associated Java object is discarded. Because multiple packed objects can be using the same data, one object might become a candidate for finalization while other objects are still accessing the data. In this situation, the finalize() method would incorrectly free the data. Therefore, finalization is not enabled for packed objects.

**The PackedObject type hierarchy:**

Because packed objects are not standard Java objects, you must understand how the classes are implemented so that you can avoid programming errors.

To operate efficiently, the JVM, and in particular the Just-In-Time (JIT) compiler, demands a clear distinction between packed and non-packed object types. As such, packed objects are not standard Java objects and do not follow the same programming rules.

All packed classes must directly or indirectly extend the class com.ibm.jvm.packed.PackedObject. Packed array classes are subclasses of PackedObject. Unlike standard Java array classes, packed array classes are not implicit subclasses of Object[]. An example type hierarchy might look as follows:

- Object
  - PackedObject
    - PackedXYZ
    - PackedXYZ.Array
  - Object[]
  - PackedObject[]
  - PackedXYZ[]
The PackedObject class contains APIs that operate on packed objects, including the following APIs:

- Factory methods for creating new packed objects
- Methods to query properties of packed objects and classes

For more information about the APIs, see Application programming reference.

The class PackedObject is abstract, and is not marked with the @Packed annotation. A field of type PackedObject is never nested, but is a standard reference type field. This model can be useful because a field in a packed type can refer to another packed type, or the same type, without fully including that referenced type.

For example, consider a linked list of packed Record objects. If the next field is of type Record, the effect would be an attempt to recursively embed the class into itself. By making the next field of type PackedObject, only a reference is stored.

Similarly, an array of the abstract type, PackedObject[], contains references to packed objects, rather than the packed object data. This array can be used to store pointers to packed objects, or to build collections of packed objects. You could write, for example, a PackedVector<T extends PackedObject> class.

Casting between different packed types: unsafe casts

In some cases, you might want to view the same data in different ways. An example is the C/C++ union type. Consider this C type:

```c
typedef struct {
    char type;
    union {
        int[2] intData;       // if type == 1
        double doubleData;    // if type == 2
        char[8] byteData;     // if type == 3
    } data;
} MyData;
```

The Java language has no way to directly represent this structure. The best approximation is a packed class:

```java
@ImportPacked({
    "com/ibm/jvm/packed/types/PackedByte",
    "com/ibm/jvm/packed/types/PackedByte$Array"
})
@Packed
final public class MyData extends PackedObject {
    private byte type;
    private @Length(8) PackedByte.Array data;
}
```

You can then use the unsafe casting API provided in the PackedObject class to obtain the alternative views of the data. For example:

```java
@ImportPacked({
    "com/ibm/jvm/packed/types/PackedByte",
    "com/ibm/jvm/packed/types/PackedByte$Array",
    "com/ibm/jvm/packed/types/PackedDouble",
    "com/ibm/jvm/packed/types/PackedInt",
    "com/ibm/jvm/packed/types/PackedInt$Array"
})
@Packed
final public class MyData extends PackedObject {
    private byte type;
    private @Length(8) PackedByte.Array data;
}
```
public PackedInt.Array getIntData() {
    // return data as a PackedInt.Array of length 2
    return data.unsafeCast(PackedInt.Array.class, 2);
}

public double getDoubleData() {
    // fetch data as a PackedDouble and return the value
    PackedDouble doubleData = data.unsafeCast(PackedDouble.class);
    return doubleData.value;
}

public PackedByte.Array getByteData() {
    // data is already a PackedByte.Array
    return data;
}
}

Note: The returned objects are all views of the same data. For example, if you use the getIntData() method and the packed array elements are altered, subsequent calls to any of the methods return the altered data.

Related information: "Packed objects" on page 16

You can use packed objects to gain greater control over the layout of objects in memory. By using packed objects, your applications have greater flexibility when they work with memory structures that are not in Java code. For example, when interoperating between Java code and other languages or environments.

Using reflection on packed objects:

Reflection for packed objects differs from the standard java.lang.reflect API. IBM extensions are provided for using reflection on packed objects.

Because the layout of packed objects differs from the layout of standard objects, accessing packed objects by using the standard java.lang.reflect APIs is not always possible. IBM API extensions are available to support this function. For more information, see the API documentation: Application programming reference.

Using reflection on fields

The standard reflect API for acquiring java.lang.reflect.Field objects, for example Class.getDeclaredFields(), is altered: any non-static fields on packed classes are not returned. Instead, you can access these fields with the class com.ibm.jvm.packed.reflect.PackedField. This class provides an API to:

- Obtain field objects
- Obtain information about field objects
- Read and write to the fields

To preserve the distinction between assignment by reference, and assignment by value, you must use separate calls to store to nested fields. For more information, see Assignment semantics.

Using reflection on arrays

Packed equivalents of the java.lang.reflect.Array and java.util.Arrays APIs are provided by the com.ibm.jvm.packed.reflect.PackedArray API. As with fields, you must use separate methods to store to nested packed elements. An API is also available so that you can directly access arrays of the packed primitive data types. This API avoids obtaining a derived object from the array, and then asking for the
Direct access by using the Unsafe API

To implement the field or array APIs, you can use an alternative version of the sun.misc.Unsafe API that is aware of packed objects: com.ibm.jvm.packed.reflect.PackedUnsafe. This API is subject to the same security rules as the standard Unsafe API implementation.

Attention: Any use of the Unsafe API might cause heap corruption, unexpected behavior, or crashes.

Limitations of using java.lang.invoke with packed objects

You cannot access instance fields of packed objects by using field setter or getter method handles. You cannot use the MethodHandles.Lookup.findSetter() and MethodHandles.Lookup.findGetter() methods to obtain such method handles; if you attempt to do so, these methods throw an UnsupportedOperationException exception. You cannot create a packed array by using a constructor method handle. You cannot use the MethodHandles.Lookup.findConstructor() method to obtain such method handles; if you attempt to do so, the method throws an UnsupportedOperationException exception.

Interfacing with the JNI:

The IBM Packed JNI Extension interface is an IBM API that facilitates working with packed objects in native code.

About this task

You can work with packed objects in native code by using the Java Native Interface (JNI), but some restrictions apply, as described in “Restrictions on using the Java Native Interface with packed objects” on page 46. You can also extend the capabilities of the JNI by using the Packed JNI Extension API. This extended IBM API includes support for the following operations:

- Allocating a new instance of a packed objects or array that references native data
- Writing to a nested field
- Reading and writing elements in a packed array
- Obtaining a direct pointer to the data of a packed object or array

Use the JNI and the Packed JNI Extension API together

Procedure

To use the Packed JNI Extension API from native code, complete the following steps:

1. Include the header file JAVA_HOME/include/jnipacked.h
2. Invoke the JNI function GetEnv() with a special version constant, J9PACKED_VERSION_0_2_PRERELEASE. The GetEnv() function gets the address of a J9PackedJNIEnv * object, which is a function table that contains the extension interface functions.
3. Call extension interface functions, as required. The extension interface functions are listed in the following section: "Packed JNI Extension API reference" on page 47.
Example

```c
#include "jnipacked.h"

void Java_pkg_TestClass_testfunc(JNIEnv *env, jclass clazz, jclass packedIntClass)
{
    jint rc = 0;
    JavaVM *javaVM = NULL;
    J9PackedJNIEnv *packedEnv = NULL;
    jobject packedObject = NULL;

    /* Get the Java VM interface */
    rc = (*env)->GetJavaVM(env, &javaVM);
    if (0 != rc) {
        /* throw an exception ... */
        return;
    }

    /* Get the Packed JNI Extension interface */
    rc = (*javaVM)->GetEnv(javaVM, (void**)&packedEnv, J9PACKED_VERSION_0_2_PRERELEASE);
    if (0 != rc) {
        /* throw an exception ... */
        return;
    }

    /* Call a function in the Packed JNI Extension interface */
    packedObject = (*packedEnv)->AllocNativePackedObject(env, packedIntClass, NULL);
}
```

For more examples, see “The Packed JNI extension: example code” on page 60.

Enabling packed object support

Packed object support is not enabled by default, and support is limited to certain garbage collection policies.

By default the JVM uses standard behavior, and packed object support is not enabled. To enable packed object support, specify the following option on the command line: `-XX:+PackedObject`

Restrictions

The following garbage collection policies are supported when packed object support is enabled:

- `gencon`
- `optthruput`
- `optavgpause`

The balanced and metronome garbage collection policies are not supported.

For 64-bit JVMs, the JIT compiler does not support packed objects in compressed references mode. If packed object support is enabled, compressed references mode is turned off. For more information about compressed references, see “Compressed references” on page 73.

Packed objects problem determination

The restrictions on the use of packed objects can cause programming mistakes that lead to errors or undesired behavior.
Common mistakes

@Packed array class "..." cannot have any fields
If this error is reported for a packed array class that does not appear to declare any fields, the error might indicate that the packed array class is not declared static. For more information, see “Packed array classes” on page 31.

Direct assignment to nested fields or packed array elements
You cannot assign by reference on a nested field or element. Use the copyFrom() method to assign by value instead. For more information, see Assignment semantics.

Missing or incorrect @ImportPacked annotation
You must list all referenced packed classes in the import annotation. Imports are never automatic; for example, an inner class does not automatically import the outer class, or the other way around. You must specify the classes with the correct, fully qualified name. For more information, see “Use of packed types in other classes” on page 22.

Comparing derived packed objects by using ‘==’ or ‘!=’
Because derived objects are created by the JVM when needed, multiple accesses to the same nested field or array element might not return an identical derived object. In particular, the test a.b == a.b, where b is a nested field in a, returns false in most cases.

Verifier errors
When a class violates one or more of the restrictions on packed code, a VerifyError exception is thrown. The associated message describes the class that caused the problem and, if applicable, the method.

When a problem is found in a method, an offset into the bytecode is provided. If you are familiar with the bytecode output of the Java compiler, you can use this offset information to locate the cause of the problem.

Related information:
“Packed objects” on page 16
You can use packed objects to gain greater control over the layout of objects in memory. By using packed objects, your applications have greater flexibility when they work with memory structures that are not in Java code. For example, when interoperating between Java code and other languages or environments.

Reference
The following reference topics include information about terminology, APIs, restrictions, example code.
• “Packed objects glossary”
• Packed objects API documentation
• “Packed objects sample package” on page 45
• “Restrictions on using the Java Native Interface with packed objects” on page 46
• “Packed JNI Extension API reference” on page 47
• “The Packed JNI extension: example code” on page 60

Packed objects glossary:
Glossary for the packed object support.
### By-reference
Indicates that an operation is performed by using references (or pointers) to the operand. For example: an assignment-by-reference assigns a reference to the source object into the destination, rather than copying the data that is contained within the object. All standard Java operations that involve objects are done by-reference.

### By-value
Indicates that an operation is performed by using the contents of the operand. For example: an assignment-by-value copies the data that is contained within the source into the destination, rather than changing a reference to point to the source. All standard Java operations that involve primitive types are done by-value.

### Derived
A small object that is created by the JVM to allow access to a nested packed object. A derived object does not have its own data; it represents a view of the data in the object from which it was derived.

### Nested
Indicates that a type is embedded within a container. A field of a packed type within another packed type is considered to be nested, as is an element in a packed array.

### Mixed
Indicates that a packed type contains one or more reference fields, or that a packed type contains one or more nested fields that are mixed packed type. A mixed object is bound by additional restrictions because the garbage collector must be able to interact with the object's data.

### Native
Describes a packed object whose data is stored outside the Java heap. Also known as off-heap.

### Off-heap
Describes a packed object whose data is stored outside the Java heap. Also known as native.

### On-heap
Describes a packed object whose data is stored within the Java heap.

### Packed
Describes a type that is marked with the @Packed annotation, or a packed array of such a type.

### Pure
Indicates that a packed type does not contain any reference fields, and does not contain any nested fields that are mixed packed type.

### Related information:
"Packed objects" on page 16
You can use packed objects to gain greater control over the layout of objects in memory. By using packed objects, your applications have greater flexibility when they work with memory structures that are not in Java code. For example, when interoperating between Java code and other languages or environments.

### Packed objects sample package:
The sample package contains classes and source code that demonstrate how to use packed objects.

The classes include simple examples and complete applications. Download the com.ibm.packedexample.zip file and extract it to a directory of your choice.
See the readme.txt file for more information and instructions for running the classes.

**Related information:**

[[“Packed objects” on page 16](#)]

You can use packed objects to gain greater control over the layout of objects in memory. By using packed objects, your applications have greater flexibility when they work with memory structures that are not in Java code. For example, when interoperating between Java code and other languages or environments.

**Packed objects API documentation**

**Restrictions on using the Java Native Interface with packed objects:**

Some Java Native Interface (JNI) functions cannot be used with packed objects, or might behave differently when used with packed objects.

**JNI functions that cannot be used with packed objects**

**GetPrimitiveArrayCritical() and ReleasePrimitiveArrayCritical()**

Packed arrays are not primitive arrays. You cannot call either of these functions on packed arrays. If you call these functions with a packed array argument, the functions fail silently, and might corrupt memory.

**MonitorEnter()**

You cannot use packed objects as object monitors. If the object argument is a PackedObject instance, this method returns -1 and fails.

**MonitorExit()**

You cannot use packed objects as object monitors. If the object argument is a PackedObject instance, this method returns -1 and throws an IllegalArgumentException exception.

**SetObjectArrayElement()**

You cannot use this function to assign the value of an element of a packed array. Use the SetPackedArrayElement() Packed JNI Extension function instead.

If you attempt to call this function on a packed array, the function fails silently. However, you can use this function to assign elements into an array of references to PackedObject instances.

**SetObjectField()**

You cannot use this function to assign the value of a nested field. Use the SetNestedPackedField() Packed JNI Extension function instead.

If you attempt to call this function on a packed array, the function fails silently.

**ToReflectedField()**

You cannot use this function to convert a jfieldID that represents an instance field of a packed class. If the incoming jfieldID argument represents such a field, this function returns NULL.

**ToReflectedMethod()**

You cannot use this function to convert a jmethodID that represents a method or constructor of a packed array class. If the incoming jmethodID argument represents such a method or constructor, this function returns NULL.
JNI functions that might behave differently when used with packed objects

GetEnv()

Use this API function to get a pointer to the Packed JNI Extensions Interface. Use a version value of J9PACKED_VERSION_0_2_PRERELEASE to get the pointer. If J9PACKED_VERSION_0_2_PRERELEASE is requested, and packed object support is not run-time-enabled, this function returns JNI_EVERSION and sets *env to NULL.

GetBooleanField(), GetByteField(), GetShortField(), GetCharField(), GetIntField(), GetLongField(), GetFloatField(), GetDoubleField(), SetBooleanField(), SetByteField(), SetShortField(), SetCharField(), SetIntField(), SetLongField(), SetFloatField(), SetDoubleField()

For a packed object, you can use these functions to get or set a field of standard primitive data type, such as boolean or int, but not a field of packed primitive data type, for example PackedBoolean or PackedInt.

If you use these functions on an incorrect type of field, they fail silently. For example, consider the following packed class C:

```java
@ImportPacked({"com/ibm/jvm/packed/types/PackedInt"})
@Packed
final class C extends PackedObject {
    int primitiveInt;
    PackedInt packedInt;
}
```

C cInst = new C();

You can get or set the cInst.primitiveInt field by using the GetIntField() and SetIntField() functions. You cannot get or set the cInst.packedInt field by using these methods because this field is of type PackedInt. Use the GetObjectField() JNI function and the SetNestedPackedField() Packed JNI extension function instead.

IsSameObject()

When the input arguments to this function are two references to the same nested field, or two references to the same packed array element, the return value is undefined. The Packed JNI Extension function IsIdentical() might be more useful in such situations.

Related information:

"Packed objects" on page 16

You can use packed objects to gain greater control over the layout of objects in memory. By using packed objects, your applications have greater flexibility when they work with memory structures that are not in Java code. For example, when interoperaing between Java code and other languages or environments.

Packed JNI Extension API reference:

This API provides functions for working with packed objects in native code.

For information about using these functions in your code, see "Interfacing with the JNI" on page 42.

Refer to the subtopics in this section for information about specific functions.

Data types

The following data types are defined in the jnipacked.h header file.
J9PackedJNIEnv

This data type is the type of the function table that contains the packed JNI extension functions.

jpackedarray

This data type represents a JNI reference to a packed array.

Related information:

“Packed objects” on page 16
You can use packed objects to gain greater control over the layout of objects in memory. By using packed objects, your applications have greater flexibility when they work with memory structures that are not in Java code. For example, when interoperating between Java code and other languages or environments.

Packed JNI Extension API: functions for accessing fields of packed objects:

You can set a nested field in a packed object.

Set a nested field

void SetNestedPackedField(JNIEnv *env, jobject packedObject, jfieldID fieldID, jobject value)

Assigns the value of a nested field. This function is analogous to the following Java code:

`packedObject.field.copyFrom(value)`

Parameters

- `env`: The JNI context. This parameter must not be `NULL`.
- `packedObject`: The packed object. This parameter must not be `NULL`.
- `fieldID`: The nested field of the packed object to assign. This parameter can be a nested packed array field, but must not be `NULL` or a `static` field.
- `value`: A packed object that represents the value to assign to the field. This parameter must not be `NULL`.

Exceptions

- `NullPointerException`, if the `value` parameter is `NULL`

Related information:

“Packed objects” on page 16
You can use packed objects to gain greater control over the layout of objects in memory. By using packed objects, your applications have greater flexibility when they work with memory structures that are not in Java code. For example, when interoperating between Java code and other languages or environments.

“Packed JNI Extension API: functions for accessing elements of packed arrays” on page 49
You can get or set an element of a packed array, using either a packed object or a specific type such as `boolean` or `int`.

“Packed JNI Extension API: functions for allocating and freeing packed objects” on page 51
You can allocate or free memory for an off-heap packed object, and you can allocate memory for on-heap and off-heap packed arrays.

“Packed JNI Extension API: functions for direct memory access” on page 54
Use these functions to operate on packed object data by using direct pointers to native memory. These functions might use data copying in some situations only; to make your code more portable, do not assume that data will be copied.

“Packed JNI Extension API: packed class utility functions” on page 58
You can get the size of the packed data for a packed class. You can also get a
packed class, or the component type of a packed class.

"Packed JNI Extension API: packed object utility functions" on page 59

The utility functions include testing whether references are identical and returning the number of elements in a packed array.

Packed JNI Extension API: functions for accessing elements of packed arrays:

You can get or set an element of a packed array, using either a packed object or a specific type such as boolean or int.

Set a packed array element

```c
void SetPackedArrayElement(JNIEnv *env, jpackedarray array, jsize index, jobject value)
```

Assigns the value of a packed array element. This function is analogous to the following Java code:

```java
array.at(index).copyFrom(value)
```

This function cannot assign data to an element of a standard array. For example, consider the following standard array:

```java
PackedObject[] refArray = new PackedObject[3];
```

The following code is not correct because `refArray` is an array of references:

```java
(*packedEnv)->SetPackedArrayElement(env, refArray, 0, value);
```

Instead, update `refArray` by using the standard JNI function `SetObjectArrayElement()`:

```java
(*env)->SetObjectArrayElement(env, refArray, 0, value);
```

Parameters

- `env`: The JNI context. This parameter must not be NULL.
- `array`: The packed array. This parameter must not be NULL.
- `index`: The index of the array element.
- `value`: A packed object that represents the value to assign into the array. This parameter must not be NULL.

Exceptions

- `IllegalArgumentException`, if the array is not a packed array
- `ArrayIndexOutOfBoundsException`, if the index is not a valid array index
- `ArrayStoreException`, if the class of the value parameter does not match the element class of the array
- `NullPointerException`, if the value parameter is NULL

Set a packed array element of a specific type

```c
void SetPackedBooleanArrayElement(JNIEnv *env, jpackedarray array, jsize index, jboolean value)
void SetPackedByteArrayElement(JNIEnv *env, jpackedarray array, jsize index, jbyte value)
void SetPackedShortArrayElement(JNIEnv *env, jpackedarray array, jsize index, jshort value)
void SetPackedCharArrayElement(JNIEnv *env, jpackedarray array, jsize index, jchar value)
void SetPackedIntArrayElement(JNIEnv *env, jpackedarray array, jsize index, jint value)
void SetPackedLongArrayElement(JNIEnv *env, jpackedarray array, jsize index, jlong value)
void SetPackedFloatArrayElement(JNIEnv *env, jpackedarray array, jsize index, jfloat value)
void SetPackedDoubleArrayElement(JNIEnv *env, jpackedarray array, jsize index, jdouble value)
```
These functions assign the value of a PackedPrimitiveType array element, where PrimitiveType is one of the following types: Boolean, Byte, Short, Char, Int, Long, Float, or Double. These functions are analogous to the following Java code:

```java
array.at(index).value = value;
```

For example, use the following code to set the value of an element from an array of PackedInt objects:

```c
(*packedEnv)->SetPackedIntArrayElement(env, array, 0, 3);
```

**Parameters**
- `env`: The JNI context. This parameter must not be NULL.
- `array`: The packed array. This parameter must not be NULL.
- `index`: The index of the array element.
- `value`: The value to assign to the array, at the position that is represented by the index parameter.

**Exceptions**
- `IllegalArgumentException`, if the array is not a packed array
- `ArrayIndexOutOfBoundsException`, if the index is not a valid array index

Get a packed array element

```c
jobject GetPackedArrayElement(JNIEnv *env, jpackedarray array, jsize index)
```

This function gets a packed array element. This function is analogous to the following Java code:

```java
array.at(index)
```

**Parameters**
- `env`: The JNI context. This parameter must not be NULL.
- `array`: The packed array. This parameter must not be NULL.
- `index`: The index of the array element.

**Returns**
An element of the packed array, or NULL if an error occurs.

**Exceptions**
- `IllegalArgumentException`, if the array is not a packed array
- `ArrayIndexOutOfBoundsException`, if the index is not a valid array index.

Get the value of a packed array element of a specific type

```c
jboolean GetPackedBooleanArrayElement(JNIEnv *env, jpackedarray array, jsize index)
jbyte GetPackedByteArrayElement(JNIEnv *env, jpackedarray array, jsize index)
jshort GetPackedShortArrayElement(JNIEnv *env, jpackedarray array, jsize index)
jchar GetPackedCharArrayElement(JNIEnv *env, jpackedarray array, jsize index)
jint GetPackedIntArrayElement(JNIEnv *env, jpackedarray array, jsize index)
jlong GetPackedLongArrayElement(JNIEnv *env, jpackedarray array, jsize index)
jfloat GetPackedFloatArrayElement(JNIEnv *env, jpackedarray array, jsize index)
jdouble GetPackedDoubleArrayElement(JNIEnv *env, jpackedarray array, jsize index)
```

These functions obtain the value of a PackedPrimitiveType array element, where PrimitiveType is one of the following types: Boolean, Byte, Short, Char, Int, Long, Float, or Double. These functions are analogous to the following Java code:

```java
array.at(index).value
```
For example, use the following code to get the value of an element from an array of PackedInt objects:

```java
jint value = (*packedEnv)->GetPackedIntArrayElement(env, array, 0);
```

**Parameters**
- `env`: The JNI context. This parameter must not be `NULL`.
- `array`: The packed array. This parameter must not be `NULL`.
- `index`: The index of the array element.

**Returns**
- The value of `array.at(index)`

**Exceptions**
- `IllegalArgumentException`, if the array is not a packed array
- `ArrayIndexOutOfBoundsException`, if the index is not a valid array index

**Related information:**
- “Packed objects” on page 16
  You can use packed objects to gain greater control over the layout of objects in memory. By using packed objects, your applications have greater flexibility when they work with memory structures that are not in Java code. For example, when interoperating between Java code and other languages or environments.

- “Packed JNI Extension API: functions for accessing fields of packed objects” on page 48
  You can set a nested field in a packed object.

- “Packed JNI Extension API: functions for allocating and freeing packed objects”
  You can allocate or free memory for an off-heap packed object, and you can allocate memory for on-heap and off-heap packed arrays.

- “Packed JNI Extension API: functions for direct memory access” on page 54
  Use these functions to operate on packed object data by using direct pointers to native memory. These functions might use data copying in some situations only; to make your code more portable, do not assume that data will be copied.

- “Packed JNI Extension API: packed class utility functions” on page 58
  You can get the size of the packed data for a packed class. You can also get a packed class, or the component type of a packed class.

- “Packed JNI Extension API: packed object utility functions” on page 59
  The utility functions include testing whether references are identical and returning the number of elements in a packed array.

**Packed JNI Extension API: functions for allocating and freeing packed objects:**

You can allocate or free memory for an off-heap packed object, and you can allocate memory for on-heap and off-heap packed arrays.

**Allocate memory for an off-heap packed object**

```java
jobject AllocNativePackedObject(JNIEnv *env, jclass clazz, void *address)
```

Allocates memory for an off-heap packed object. None of the constructors for the object are called.

If the address is not `NULL`, the address refers to preallocated memory that is wrapped by the returned packed object. To determine the amount of memory to preallocate, use the `GetClassPackedDataSize()` function.
If the address is NULL, the JVM allocates native memory for the packed object. Use the FreeNativePackedObject() function to free the native memory.

**Parameters**
- env: The JNI context. This parameter must not be NULL.
- clazz: A final, pure packed, non-array class.
- address: Either a pointer to preallocated memory, or NULL.

**Returns**
- A new packed object of class clazz, or NULL if the function fails.

**Exceptions**
- OutOfMemoryError, if the function fails to allocate heap or native memory.
- IllegalArgumentException, if the clazz parameter is not a packed class, or it is a mixed packed class
- InstantiationException, if the clazz parameter is an abstract packed class or a packed array class
- NullPointerException, if the clazz parameter is NULL

Allocate memory for a packed array

```
jpackedarray AllocPackedArray(JNIEnv *env, jclass elementClass, jsize length)
```

Allocates an on-heap packed array. No constructors for the array elements are called.

**Parameters**
- env: The JNI context. This parameter cannot be NULL.
- elementClass: A final, packed, non-array class.
- length: The number of array elements. This parameter can be 0.

**Returns**
- A new packed array of objects of class elementClass, or NULL if the function fails.

**Exceptions**
- ClassNotFoundException, if the packed array class that corresponds to the elementClass class is not defined. For more information, see "Packed array classes" on page 31.
- IllegalArgumentException, if the elementClass parameter is not a packed class, or if it is a packed array class
- InstantiationException, if the elementClass parameter is an abstract class
- NullPointerException, if the elementClass parameter is NULL
- NegativeArraySizeException, if the length parameter is less than 0
- OutOfMemoryError, if the function fails to allocate heap or native memory

Allocate memory for an off-heap packed array

```
jpackedarray AllocNativePackedArray(JNIEnv *env, jclass elementClass, jsize length, void *address)
```

Allocates an off-heap packed array. No constructors for the array elements are called.
If the address is not NULL, the address refers to preallocated memory that is wrapped by the returned packed array. To determine the amount of memory to preallocate, use the GetClassPackedDataSize() function, and multiply the result by the packed array length.

If the address is NULL, the JVM allocates native memory for the packed array. Use the FreeNativePackedObject() function to release the native memory.

**Parameters**

- env: The JNI context. This parameter must not be NULL.
- elementClass: A final, pure packed, non-array class.
- length: The number of array elements. This parameter can be 0.
- address: Either a pointer to preallocated memory, or NULL.

**Returns**

- A new packed array of objects of class elementClass, or NULL if the function fails.

**Exceptions**

- ClassNotFoundException, if the packed array class that corresponds to the elementClass class is not defined. For more information, see “Packed array classes” on page 31.
- OutOfMemoryError, if the function fails to allocate heap or native memory
- IllegalArgumentException, if the elementClass parameter is not a packed class, or if it is a mixed packed class or a packed array class
- InstantiationException, if the elementClass parameter is an abstract class
- NullPointerException, if the elementClass parameter is NULL
- NegativeArraySizeException, if the length parameter is less than 0

**Frees an off-heap packed object**

```c
void FreeNativePackedObject(JNIEnv *env, jobject nativePackedObject)
```

Frees the native memory for a JVM-allocated off-heap packed object. Call this function only on off-heap packed objects, or off-heap packed arrays, that were allocated by the JVM (that is, by using the AllocNativePackedObject(env, clazz, NULL) or AllocNativePackedArray(env, elementClass, length, NULL) functions).

**Attention:** If the packed object was not allocated by the JVM, this function might corrupt memory.

**Parameters**

- env: The JNI context. This parameter must not be NULL.
- nativePackedObject: A JVM-allocated native packed object. This object can be a singleton or an array object.

**Exceptions**

- IllegalArgumentException, if the nativePackedObject parameter is not a packed object
- NullPointerException, if the nativePackedObject parameter is NULL

**Related information:**
You can use packed objects to gain greater control over the layout of objects in memory. By using packed objects, your applications have greater flexibility when they work with memory structures that are not in Java code. For example, when interoperating between Java code and other languages or environments.

You can set a nested field in a packed object.

You can get or set an element of a packed array, using either a packed object or a specific type such as boolean or int.

Use these functions to operate on packed object data by using direct pointers to native memory. These functions might use data copying in some situations only; to make your code more portable, do not assume that data will be copied.

You can get the size of the packed data for a packed class. You can also get a packed class, or the component type of a packed class.

The utility functions include testing whether references are identical and returning the number of elements in a packed array.

**Get a pointer to a packed object**

```c
void *GetPackedObjectPointer(JNIEnv *env, jobject packedObject, jboolean *isCopy)
```

Returns a direct pointer to the body of a packed object. The result is valid until the ReleasePackedObjectPointer() function is called. The returned pointer might refer either to the original packed object, or to a copy of it. Changes that you make are not necessarily reflected in the original packed object until ReleasePackedObjectPointer() is called.

You cannot use this function on a mixed packed object.

**Parameters**

- `env`: The JNI context. This parameter must not be NULL.
- `packedObject`: The packed object to access. This parameter must not be NULL.
- `isCopy`: This parameter is an input and an output parameter. If the input `isCopy` parameter is not NULL, this function sets `isCopy` to JNI_TRUE if a copy is made, or JNI_FALSE if no copy is made.

**Returns**

- A pointer to the body of the packed object, or NULL if the operation failed. Some possible reasons for failure are as follows:
  - The packed object is NULL
  - The packed object is a mixed packed object
  - Memory could not be allocated
Exceptions

- OutOfMemoryError, if memory could not be allocated
- IllegalArgumentException, if the packed object is not a pure packed object

Release a pointer to a packed object

```c
void ReleasePackedObjectPointer(JNIEnv *env, jobject packedObject, void *ptr, jint mode)
```

Releases a pointer that was obtained by using the GetPackedObjectPointer() function. This function can also commit changes to the original packed object.

For the `mode` parameter, use one of the values listed in the following table.

<table>
<thead>
<tr>
<th>Table 3. Possible values for the mode parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>JNI_COMMIT</td>
</tr>
<tr>
<td>JNI_ABORT</td>
</tr>
</tbody>
</table>

You cannot use this function on a mixed packed object.

Parameters

- `env`: The JNI context. This parameter must not be NULL.
- `packedObject`: The packed object to access. This parameter must not be NULL.
- `ptr`: The pointer that was obtained by using the GetPackedObjectPointer() function.
- `mode`: Possible values are as follows: 0, JNI_COMMIT, or JNI_ABORT.

Exceptions

- IllegalArgumentException, if the packed object is not a pure packed object

Get packed array elements

```c
void *GetPackedArrayElements(JNIEnv *env, jpackedarray packedArray, jboolean *isCopy)
```

Returns a direct pointer to elements of a packed array. The result is valid until the ReleasePackedArrayElements() function is called. The returned pointer might refer either to the original array elements, or to a copy of them. Changes that you make to the elements are not necessarily reflected in the original packed array until the ReleasePackedArrayElements() function is called.

You cannot use this function on a mixed packed array.

Parameters

- `env`: The JNI context. This parameter must not be NULL.
- `packedArray`: The packed array to access. This parameter must not be NULL.
- isCopy: If the isCopy parameter is not NULL, this function sets *isCopy to JNI_TRUE if a copy is made, or JNI_FALSE if no copy is made.

**Returns**
- A pointer to the elements of a packed array, or NULL if the operation failed. Some possible reasons for failure are as follows:
  - The packed array is NULL
  - The packed array is a mixed packed object
  - Memory could not be allocated

**Exceptions**
- OutOfMemoryError, if memory could not be allocated
- IllegalArgumentException, if the packedArray parameter is not a pure packed array

**Release a pointer to packed array elements**

```c
void ReleasePackedArrayElements(JNIEnv *env, jpackedarray packedArray, void *elems, jint mode)
```

Releases a pointer that was obtained by using the GetPackedArrayElements() function. This function can also commit changes to the original packed array.

You cannot use this function on a mixed packed array.

**Parameters**
- env: The JNI context. This parameter must not be NULL.
- packedArray: The packed array to access. This parameter must not be NULL.
- ptr: The pointer that was obtained by using the GetPackedArrayElements() function.
- mode: Possible values are as follows: 0, JNI_COMMIT, or JNI_ABORT. See ReleasePackedObjectPointer() for a description of the mode values.

**Exceptions**
- IllegalArgumentException, if the packedArray parameter is not a pure packed array

**Get part of a packed array**

```c
void GetPackedArrayRegion(JNIEnv *env, jpackedarray packedArray, jsize start, jsize length, void *buf)
```

Copies part of a packed array into a preallocated native buffer.

You cannot use this function on a mixed packed array.

The required minimum size of the native buffer is the length of the packed array region multiplied by the packed data size of the packed array’s class. Use the GetClassPackedDataSize() function to get the packed data size.

**Parameters**
- env: The JNI context. This parameter must not be NULL.
- packedArray: The packed array. The array must not be a mixed packed array or NULL.
- start: The first array element to copy.
- length: The number of array elements to copy. This parameter must be 0 or greater.
buf: A preallocated native buffer. This parameter is an input and an output parameter.

Exceptions

- IllegalArgumentException, if the packed array is a mixed packed array
- ArrayIndexOutOfBoundsException, if the \([\text{start}, \text{start}+\text{length}-1]\) values are not all valid indexes for the array
- NullPointerException, if the \(\text{buf}\) parameter is \text{NULL}

Set part of a packed array

void SetPackedArrayRegion(JNIEnv* env, jpackedarray packedArray, jsize start, jsize length, void* buf)

Copies from a native buffer into part of a packed array.

You cannot use this function on a mixed packed array.

Parameters

- env: The JNI context. This parameter must not be \text{NULL}.
- packedArray: The packed array. The array must not be a mixed packed array or \text{NULL}.
- start: The first array element to copy.
- length: The number of array elements to copy.
- buf: A preallocated native buffer.

Exceptions

- IllegalArgumentException, if the packed array is a mixed packed array
- ArrayIndexOutOfBoundsException, if the \([\text{start}, \text{start}+\text{length}-1]\) values are not all valid indexes for the array
- NullPointerException, if the \(\text{buf}\) parameter is \text{NULL}

Related information:

“Packed objects” on page 16
You can use packed objects to gain greater control over the layout of objects in memory. By using packed objects, your applications have greater flexibility when they work with memory structures that are not in Java code. For example, when interoperating between Java code and other languages or environments.

“Packed JNI Extension API: functions for accessing fields of packed objects” on page 48
You can set a nested field in a packed object.

“Packed JNI Extension API: functions for accessing elements of packed arrays” on page 49
You can get or set an element of a packed array, using either a packed object or a specific type such as \text{boolean} or \text{int}.

“Packed JNI Extension API: functions for allocating and freeing packed objects” on page 51
You can allocate or free memory for an off-heap packed object, and you can allocate memory for on-heap and off-heap packed arrays.

“Packed JNI Extension API: packed class utility functions” on page 58
You can get the size of the packed data for a packed class. You can also get a packed class, or the component type of a packed class.

“Packed JNI Extension API: packed object utility functions” on page 59
The utility functions include testing whether references are identical and returning the number of elements in a packed array.
Packed JNI Extension API: packed class utility functions:

You can get the size of the packed data for a packed class. You can also get a packed class, or the component type of a packed class.

Get the size of the packed data for a non-array packed class

```java
jlong GetClassPackedDataSize(JNIEnv *env, jclass packedClass)
```

Returns the size of the packed data for a packed class. For a packed array, this function returns the size of one element.

**Parameters**
- `env`: The JNI context. This parameter must not be `NULL`.
- `packedClass`: A packed class. This parameter must not be `NULL`.

**Returns**
- The size, in bytes, that is required for the packed data in an instance of the packed class, or -1 if an error occurs. An error occurs if `packedClass` is not a packed class.

Get a packed array class

```java
jclass GetPackedArrayClass(JNIEnv *env, jclass elementClass)
```

Returns the packed array class for the specified element class.

The element class must be final.

The element class must not be a packed array class. Only one-dimensional packed arrays are supported.

**Parameters**
- `env`: The JNI context. This parameter must not be `NULL`.
- `elementClass`: A packed non-array class. This parameter must not be `NULL`.

**Returns**
- Returns a packed array class, or `NULL` if an error occurs.

**Exceptions**
- `ClassNotFoundException`, if the packed array class definition cannot be found
- `IllegalArgumentException`, if the `elementClass` parameter is not a packed class, or if it is abstract or a packed array class
- `OutOfMemoryError`, if the function fails to allocate native or heap memory

Get the component type of a packed array class

```java
jclass GetPackedArrayClassComponentType(JNIEnv *env, jclass packedArrayClass)
```

Returns the component type of a packed array class.

**Parameters**
- `env`: The JNI context. This parameter must not be `NULL`.
- `packedArrayClass`: A packed array class. This parameter must not be `NULL`.
Returns

- Returns the component type of the packed array class, or NULL if an error occurs.

Related information:

“Packed objects” on page 16
You can use packed objects to gain greater control over the layout of objects in memory. By using packed objects, your applications have greater flexibility when they work with memory structures that are not in Java code. For example, when interoperating between Java code and other languages or environments.

“Packed JNI Extension API: functions for accessing fields of packed objects” on page 48
You can set a nested field in a packed object.

“Packed JNI Extension API: functions for accessing elements of packed arrays” on page 49
You can get or set an element of a packed array, using either a packed object or a specific type such as boolean or int.

“Packed JNI Extension API: functions for allocating and freeing packed objects” on page 51
You can allocate or free memory for an off-heap packed object, and you can allocate memory for on-heap and off-heap packed arrays.

“Packed JNI Extension API: functions for direct memory access” on page 54
Use these functions to operate on packed object data by using direct pointers to native memory. These functions might use data copying in some situations only; to make your code more portable, do not assume that data will be copied.

“Packed JNI Extension API: packed object utility functions”
The utility functions include testing whether references are identical and returning the number of elements in a packed array.

Packed JNI Extension API: packed object utility functions:

The utility functions include testing whether references are identical and returning the number of elements in a packed array.

Tests whether two references are identical

jboolean IsIdentical(JNIEnv *env, jobject ref1, jobject ref2)

Tests whether two references are identical.

For non-packed objects, this function is equivalent to the standard JNI function IsSameObject().

For packed objects, two references are identical if all the following conditions apply:
- The references both reference the same packed object data
- Both of the objects have the same type
- In the case of packed arrays, the objects both have the same packed array length

Parameters

- env: The JNI context. This parameter must not be NULL.
- ref1: A Java object.
- ref2: A Java object.
Returns
Returns JNI_TRUE if the two objects are identical, otherwise JNI_FALSE.

Get the number of elements in a packed array

jsize GetPackedArrayLength(JNIEnv *env, jpackedarray array)

Returns the number of elements in a packed array.

Parameters
• env: The JNI context. This parameter must not be NULL.
• array: A packed array. This parameter must not be NULL.

Returns
Returns the number of elements in the packed array.

Related information:
“Packed objects” on page 16
You can use packed objects to gain greater control over the layout of objects in memory. By using packed objects, your applications have greater flexibility when they work with memory structures that are not in Java code. For example, when interoping between Java code and other languages or environments.

“Packed JNI Extension API: functions for accessing fields of packed objects” on page 48
You can set a nested field in a packed object.

“Packed JNI Extension API: functions for accessing elements of packed arrays” on page 49
You can get or set an element of a packed array, using either a packed object or a specific type such as boolean or int.

“Packed JNI Extension API: functions for allocating and freeing packed objects” on page 51
You can allocate or free memory for an off-heap packed object, and you can allocate memory for on-heap and off-heap packed arrays.

“Packed JNI Extension API: functions for direct memory access” on page 54
Use these functions to operate on packed object data by using direct pointers to native memory. These functions might use data copying in some situations only; to make your code more portable, do not assume that data will be copied.

“Packed JNI Extension API: packed class utility functions” on page 58
You can get the size of the packed data for a packed class. You can also get a packed class, or the component type of a packed class.

The Packed JNI extension: example code:

Examples of how to use the packed JNI extension to work with native objects.

Assume you define the following Java data structures:

PackedPixel.java

import java.io.Serializable;
import com.ibm.jvm.packed.ImportPacked;
import com.ibm.jvm.packed.Packed;
import com.ibm.jvm.packed.PackedObject;
import com.ibm.jvm.packed.reflect.PackedArray;

@Packed
public final class PackedPixel extends PackedObject {
    byte r;
    byte g;
byte b;
byte a;

@ImportPacked({"PackedPixel"})
@Packed
class Array extends PackedObject implements Serializable, Cloneable {
  private Array() {
  }

  public static Array allocate(int length) {
    return PackedArray.newArray(Array.class, length);
  }

  public PackedPixel at(int index) {
    return PackedArray.at(this, index);
  }

  public int getLength() {
    return PackedArray.getLength(this);
  }

  public Array clone() throws CloneNotSupportedException {
    return (Array)super.clone();
  }
}

PackedBitmap8x8.java

import com.ibm.jvm.packed.ImportPacked;
import com.ibm.jvm.packed.Length;
import com.ibm.jvm.packed.Packed;
import com.ibm.jvm.packed.PackedObject;

@ImportPacked({"PackedPixel", "PackedPixel$Array"})
@Packed
final class PackedBitmap8x8 extends PackedObject {
  @Length(64) PackedPixel.Array data;
}

In C code, you could define a data structure that has the same memory layout as
the PackedPixel class, as follows:

#include "jni.h"
#include "jnipacked.h"

typedef struct CPackedPixel {
  jbyte r;
  jbyte g;
  jbyte b;
  jbyte a;
} CPackedPixel;

The following examples show how you can manipulate PackedPixel and
PackedBitmap8x8 objects in C code. Error checking has been omitted for clarity.

jint rc = JNI_OK;
JavaVM *vm = NULL;
J9PackedJNIEnv *packedEnv = NULL;
jclass bmpClass = NULL;
jobject bmpObj = NULL;
jfieldID dataFieldID = NULL;
jpackedarray dataField = NULL;
CPackedPixel *ppix = NULL;

/* Get the Java VM interface */
rc = (*env)->GetJavaVM(env, &vm);
/* Get the Packed JNI extension */
rc = (*vm)->GetEnv(vm, (void**)&packedEnv, J9PACKED_VERSION_0_2_PRERELEASE);

/* Get the PackedBitmap8x8 class */
bmpClass = (*env)->FindClass(env, "LPackedBitmap8x8;");

/* Get the field ID of the "data" field of the PackedBitmap8x8 class */
dataFieldID = (*env)->GetFieldID(env, bmpClass, "data", "[LPackedPixel$Array;";

/* Allocate an off-heap instance of the PackedBitmap8x8 class */
bmpObj = (*packedEnv)->AllocNativePackedObject(env, bmpClass, NULL);

/* Access the "data" field of the PackedBitmap8x8 instance */
dataField = (jpixedarray)(*env)->GetObjectField(env, bmpObj, dataFieldID);

/* Get a direct pointer to the array elements of the data field */
ppix = (CPackedPixel *)(*packedEnv)->GetPackedArrayElements(env, dataField, NULL);

/* Set the values of an array element */
ppix[0].r = 50;
ppix[0].g = 80;
ppix[0].b = 0;
ppix[0].a = 10;

/* Commit the changes back to the original data field */
(*packedEnv)->ReleasePackedArrayElements(env, dataField, ppix, 0);

/* Free the instance of the PackedBitmap8x8 class */
(*packedEnv)->FreeNativePackedObject(env, bmpObj);

Related tasks:
"Interfacing with the JNI" on page 42
The IBM Packed JNI Extension interface is an IBM API that facilitates working with packed objects in native code.

Related information:
"Packed objects" on page 16
You can use packed objects to gain greater control over the layout of objects in memory. By using packed objects, your applications have greater flexibility when they work with memory structures that are not in Java code. For example, when interoperating between Java code and other languages or environments.

"Packed JNI Extension API reference" on page 47
This API provides functions for working with packed objects in native code.

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

Methods and classes are shown in normal font:
• The serviceCall() method
• The StreamRemoteCall class

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

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

-Xverify:[remote | all | none]
with the default underscored.

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

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

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

**Directory conventions**

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

**`install_dir`**

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

- AIX: `/usr/java<version><release>[_64]/`

For example:

- AIX: `/usr/java71_64/`

**`lib_dir`**

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

- AIX: `install_dir/jre/lib/ppc[64]/`

**`vm_dir`**

The virtual machine (VM) directory is referred to as `vm_dir` in this documentation. The VM directory is as follows:

- AIX: `install_dir/jre/lib/ppc[64]/default`

If you are using compressed references, the `vm_dir` points to the `compressedrefs` subdirectory, not the `default` subdirectory on the same path.

**Java virtual machine (VM) version conventions**

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

```
java -version
```

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

```
java version "1.7.0"
Java(TM) SE Runtime Environment (build pwa6470.27-20130715_03) IBM J9 VM (build 2.7, JRE 1.7.0 Windows 7 amd64-64 20130711_156087 (JIT enabled, AOT enabled) J9VM - R27_head_20130711_0404_B156087 JIT - tr.r13.java_20130709.41534 GC - R27_head_20130711_0404_B156087 J9CL - 20130711_156087) JCL - 20130704_01 based on Oracle 7u25-b12
```

The format of `<vm_version>` is digits only, so in the previous example, `<vm_version>` is 27.
Other sources of information

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

- To download the IBM SDKs, visit the

  [Java SDK developer center](https://www.ibm.com/support/docview.wss?uid=swg21639279)

- To download IBM SDK documentation as an Eclipse plug-in, or in PDF format for printing, see the "Downloadable documentation" topic alongside the user guides.

- For any late breaking information that is not in this guide, see:


- For API documentation that has been generated from the IBM SDK, see:

  [API documentation](https://www.ibm.com/support/docview.wss?uid=swg21639279)

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


- For Java documentation produced by Oracle, see:


---

Accessibility

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

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

For example, you can operate the IBM SDK, Java Technology Edition, Version 7 Release 1 without a mouse, by using only the keyboard.

Issues that affect accessibility are included in “Known issues and limitations” on page 606.

**Keyboard navigation**

This product uses standard Microsoft Windows navigation keys.

If you want to use keyboard navigation, see “Default Swing key bindings” on page 609 for a description of useful keystrokes for Swing applications.

**IBM and accessibility**

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility.
Chapter 2. Understanding the components

Gain a basic understanding of the IBM technologies that are included in this product.

The following content is included:
• Background information on IBM components of the SDK.
• Useful information for application designers.
• An explanation of some parts of the JVM.

This content is not intended as a description of the design of the SDK.

The sections in this part are:
• “The building blocks of the IBM Virtual Machine for Java”
• “Memory management” on page 69
• “Class loading” on page 101
• “Class data sharing” on page 104
• “The JIT compiler” on page 105
• “Java Remote Method Invocation” on page 110
• “The ORB” on page 112
• “The Java Native Interface (JNI)” on page 132

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 from IBM. The JVM is a virtualized computing machine that follows a well-defined specification for the runtime requirements of the Java programming language.

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

The JVM specification also defines several other runtime characteristics.

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

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

All the JVMs that are currently used commercially come with a supplementary compiler that takes bytecodes and produces platform-dependent machine code. This compiler works with the JVM to select parts of the Java program that could benefit from the compilation of bytecode, and replaces the JVM's virtualized interpretation of these areas of bytecode with concrete code. This is called just-in-time (JIT) compilation. IBM's JIT compiler is described in "The JIT compiler" on page 105.

The diagnostic information in this guide discusses the characteristics of the IBM runtime environment 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 runtime environment 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 runtime environment.
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 132).

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:

```
<table>
<thead>
<tr>
<th>JVM API</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Diagnostics</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Platform port layer</td>
</tr>
</tbody>
</table>
```

**JVM Application Programming Interface (API)**
The JVM API encapsulates all the interaction between external programs and the JVM.

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

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

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

If a problem occurs in the field, it is possible to use the capabilities of the diagnostic component to trace the runtime function of the JVM and help to identify the cause of the problem. The diagnostic component can produce output selectively from various parts of the JVM and the JIT. “Using diagnostic tools” on page 312 describes various uses of the diagnostic component.
Memory management
The memory management component is responsible for the efficient use of system memory by a Java application.

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

The memory management component has several policy options that you can specify when you deploy the application. "Memory management" on page 69 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 105.

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 71
- “Detailed description of global garbage collection” on page 74
- “Generational Concurrent Garbage Collector” on page 83
- “How to do heap sizing” on page 92
- “Interaction of the Garbage Collector with applications” on page 94
- “How to coexist with the Garbage Collector” on page 95
- “Frequently asked questions about the Garbage Collector” on page 98

For detailed information about diagnosing Garbage Collector problems, see “Garbage Collector diagnostic data” on page 436.

See also the reference information in “Garbage Collector command-line options” on page 577.

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 85.

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

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

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

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

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

For the majority of applications, the default settings work well. The heap expands until it reaches a steady state, then remains in that state, which should give a heap occupancy (the amount of live data on the heap at any given time) of 70%. At this level, the frequency and pause time of garbage collection should be acceptable.

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

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

The heap is fully expanded and the occupancy level is greater than 70%.
Increase the -Xmx value so that the heap is not more than 70% occupied. The maximum heap size should, if possible, be able to be contained in physical memory to avoid paging. For the best performance, try to ensure that the heap never pages.
At 70% occupancy the frequency of garbage collections is too great.
Change the setting of -Xminf. The default is 0.3, which tries to maintain 30% free space by expanding the heap. A setting of 0.4, for example, increases this free space target to 40%, and reduces the frequency of garbage collections.

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

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

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

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

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

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

As the name implies, heap lock allocation requires a lock and is therefore avoided, if possible, by using the cache. For a description of cache allocation, see "Cache allocation" on page 72.

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

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

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

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

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

**Large Object Area**

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

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

The Balanced Garbage Collection policy does not use the LOA. Therefore, when specifying `-Xgcpolicy:balanced`, any LOA options passed on the command line are ignored. The policy addresses the issues of LOA by reorganizing object layout with the JVM to reduce heap fragmentation and compaction requirements. This change is contained completely within the JVM, and requires no knowledge or code changes in Java.

**Initialization and the LOA:**

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

The options take values between 0 and 0.95 (0% thru 95% of the current tenure heap size). The defaults are:
- `-Xloainitial0.05` (5%)
- `-Xloaminimum0` (0%)
- `-Xloamaximum0.5` (50%)
Expansion and shrinkage of the LOA:

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

The Garbage Collector uses the following algorithm:

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

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

Allocation in the LOA:

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

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

Compressed references

When using compressed references, the virtual machine (VM) stores all references to objects, classes, threads, and monitors as 32-bit values. Use the -Xcompressedrefs and -Xnocompressedrefs command-line options to enable or disable compressed references in a 64-bit VM. These options are recognized only by 64-bit VMs.

The use of compressed references improves the performance of many applications because objects are smaller, resulting in less frequent garbage collection, and improved memory cache utilization. Certain applications might not benefit from compressed references. Test the performance of your application with and without compressed references to determine if they are appropriate. For default option settings, see “JVM command-line options” on page 549.

Using compressed references runs a different version of the IBM J9 VM. Therefore, you need to enable compressed references if you use the dump extractor to analyze dumps that are produced by the VM. For more information about the dump extractor, see “Using jextract” on page 382.
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.

The `-Xmcrs` option allows you to set an initial size for an area in memory that is reserved for compressed references within the lowest 4 GB memory area. Setting this option secures space for native classes, monitors, and threads that are used by compressed references. For more information, see “-Xmcrs” on page 587. An alternative method for securing space in the heap is to use the `-Xgc:preferredHeapBase`, which allows you to allocate the heap you specify with the `-Xmx` option, in a memory range of your choice. For more information, see “-Xgc” on page 582.

64-bit VMs 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 J9 VM, for 64-bit platforms. The options might not be supported in subsequent releases.

Related concepts:

“More effective heap usage using compressed references” on page 216

Many Java application workloads depend on the Java heap size. The IBM SDK can use compressed references on 64-bit platforms to decrease the size of Java objects and make more effective use of the available space. The result is less frequent garbage collection and improved memory cache utilization.

**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 98.

Concurrent mark:

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

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

While the GC is marking live objects concurrently with application threads running, it must record any changes to objects that are already traced. It uses a write barrier that is run every time a reference in an object is updated. The write barrier flags when an object reference update has occurred. The flag is used to force a rescan of part of the heap.
The heap is divided into 512 byte sections. Each section is allocated a single-byte card in the card table. Whenever a reference to an object is updated, the card that corresponds to the start address of the object that has been updated with the new object reference is marked with the hex value 0x01. A byte is used instead of a bit to eliminate contention, by allowing cards to be marked using non-atomic operations. A stop-the-world (STW) collection is started when one of the following events takes place:

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

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

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

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

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

The -Xgcpolicy options have these effects:

- gencon: Enables concurrent mark, and uses it in combination with generational garbage collection to help minimize the time that is spent in any garbage collection pause. gencon is the default setting. If you are having problems with erratic application response times that are caused by normal garbage collections, you can reduce those problems, reduce heap fragmentation, and still maintain good throughput, by using the gencon option. This option is particularly useful for applications that use many short-lived objects.

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

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

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

Disables concurrent mark. This policy does use concurrent garbage collection technology, but not in the way that concurrent mark is implemented here. For more information, see "Global Mark Phase" on page 88.

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 cycle of execution.

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

**Compaction phase**

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

The following analogy might help you understand the compaction process. Think of the heap as a warehouse that is partly full of pieces of furniture of different sizes. The free space is the gaps between the furniture. The free list contains only gaps that are larger than a particular size. Compaction pushes everything in one direction and closes all the gaps. It starts with the object that is closest to the wall, and puts that object against the wall. Then it takes the second object in line and puts that against the first. Then it takes the third and puts it against the second, and so on. At the end, all the furniture is at one end of the warehouse and all the free space is at the other.

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

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

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

The heap is fully expanded and less than 4% of old space is free.

Less than 128 KB of the heap is free.

With the Balanced Garbage Collection policy, the **-Xcompactgc** and **-Xnocompactgc** options are respected only if a global garbage collection is required. A global
garbage collection occurs in rare circumstances, as described in "Detailed
description of global garbage collection" on page 74. 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 577. If there is a shortage of available storage, all soft references are cleared. All soft references are guaranteed to have been cleared before the OutOfMemoryError is thrown.

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

**Final reference processing**
The processing of objects that require finalization is more straightforward.

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

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

Failure to delete a JNI Weak reference causes a memory leak in the table and performance problems. This also applies to JNI global references. The processing of JNI weak references is handled last in the reference handling process. The result is that a JNI weak reference can exist for an object that has already been finalized and had a phantom reference queued and processed.

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

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

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

The amount to expand the heap is calculated as follows:

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

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

3. If garbage collection did not free enough storage, the GC ensures that the heap
is expanded by at least the value of the allocation request.

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

**Heap shrinkage**

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

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

- The Garbage Collector (GC) did not free enough space to satisfy the allocation
  request.
- The maximum free space, which can be set by the -Xmaxf parameter (default is
  60%), is set to 100%.
- The heap has been expanded in the last three garbage collections.
- This is a System.gc() and the amount of free space at the beginning of the
garbage collection was less than -Xminf (default is 30%) of the live part of the
  heap.
- If none of the previous options are true, and more than -Xmaxf free space exists,
  the GC must calculate how much to shrink the heap to get it to -Xmaxf free
  space, without dropping to less than the initial (-Xms) value. This figure is
  rounded down to a 512-byte boundary on 32-bit JVMs or a 1024-byte boundary
  on 64-bit JVMs.

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

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

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

Whether any physical memory is released depends on the ability of the native
operating system. If it supports *paging*; the ability of the native operating system to
commit and decommit physical storage to the virtual storage; the GC uses this
function. In this case, physical memory can be decommitted on a heap shrinkage.
You never see the amount of virtual storage that is used by the JVM decrease. You might see physical memory free size increase after a heap shrinkage. The native operating system determines what it does with decommitted pages.

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

**Generational Concurrent Garbage Collector**

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 in the range 1 - 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 that is used in the new area.

You can set the initial scavenger tenure age by using the -Xgcs:cvTenureAge option. You can also turn off the adaptive tenure age with the -Xgcs:cvNoAdaptiveTenure option to ensure that the initial age set is maintained throughout the run time for a JVM. For more information, see “-Xgc” on page 582.

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:

```
0 unused 1 0 unused 2 3
```

**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.
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 \texttt{-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.

\textbf{Note:} The \texttt{-Xpartialcompactgc} option, which in previous versions of the Java runtime environment 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:

\begin{center}
\includegraphics[width=0.3\textwidth]{example_gmp.png}
\end{center}

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:

```
0  unused  1  0
```

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

```
0  unused  1  0
```

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

```
unused  1  unused  2  unused
```

**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:

```
0  2  1  0
```

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

```
0  2  1  0
```
This diagram shows the Java heap after the Mark-Compact operation, where region ages have been incremented:

![Diagram of Java heap after Mark-Compact operation]

**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 the “Concurrent mark” on page 76.

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:

![Diagram of Java heap after end of GMP]

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

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

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

Requirements

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

Performance implications

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

When to use the policy

Consider using the policy in the following situations:

The application occasionally experiences unacceptably long global garbage collection pause times

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

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

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

Other areas that might benefit

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

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

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

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

When not to use the policy

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

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

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

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

You can select the Metronome Garbage Collection policy by specifying 
-Xgcpolicy=metronome on the command line when you start your application. For example:
```
java -Xgcpolicy=metronome -Xgc:targetUtilization=80 yourApplication
```

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

Metronome Garbage Collector threads
The Metronome Garbage Collector consists of two types of threads: a single alarm thread, and a number of collection (GC) threads. By default, GC uses one thread for each logical active processor available to the operating system. This enables the most efficient parallel processing during GC cycles. A GC cycle means the time between GC being triggered and the completion of freeing garbage. Depending on the Java heap size, the elapsed time for a complete GC cycle could be several seconds. A GC cycle usually contains hundreds of GC quanta. These quanta are the very short pauses to application code, typically lasting 3 milliseconds. Use 
-verbose:gc to get summary reports of cycles and quanta. For more information, see: "Using verbose:gc information" on page 456. You can set the number of GC threads for the JVM using the -Xgcthreads option.

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

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

You cannot change the number of alarm threads for the JVM.
The Metronome Garbage Collector periodically checks the JVM to see if the heap memory has sufficient free space. When the amount of free space falls below the limit, the Metronome Garbage Collector triggers the JVM to start garbage collection.

**Alarm thread**

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

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

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

**Collection threads**

The collection threads perform the garbage collection.

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

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

For more information about using the Metronome Garbage Collector, see “Using the Metronome Garbage Collector” on page 218.

For more information about troubleshooting, see “Troubleshooting the Metronome Garbage Collector” on page 456.

**How to do heap sizing**

You can do heap sizing to suit your requirements.

If you do not set an initial or maximum heap size, the GC expands and shrinks the heap as required. This capability allows for situations where usage varies over time or exceeds an expected maximum heap size for your application. However, you can choose not to have a variable heap but to set the Java heap size for your application. There are advantages and disadvantages to setting a fixed heap and the decision to use one depends on the application you are running.

Generally:

- Do not start with a minimum heap size that is the same as the maximum heap size.
- Use a tool such as the “Garbage Collection and Memory Visualizer” on page 320 (GCMV) to check your application heap size and determine the minimum and maximum settings. Alternatively, you can use the output from `-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.

The Garbage Collector adapts heap size to keep occupancy between 40% and 70% for the following reasons:

- A heap occupancy greater than 70% causes more frequent GC cycles, which can reduce performance. You can alter this behavior by setting the `-Xminf` option.
- A heap occupancy less than 40% means infrequent GC cycles. However, these cycles are longer than necessary, causing longer pause times, which can reduce performance. You can alter this behavior by setting the `-Xmaxf` option.

If you do not set an initial or maximum heap size, the GC expands and shrinks the heap as required. However, if you fix the heap size by using the `-Xms` and `-Xmx` options, the GC does not expand or shrink the Java heap. To optimize application performance and keep within the 40 - 70% range, the maximum heap size setting should therefore be at least 43% larger than the maximum occupancy of the application. For example, if an application has a maximum occupancy of 70 MB, a maximum heap size of 100 MB should be set as shown in the following calculation:

\[ 70 + (70 \times 43/100) \]

Setting the minimum and maximum heap size to the same value is typically not a good idea because garbage collection is delayed 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. Start your application with the minimum heap size that your application requires. When the GC starts up, it runs 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 81), 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 that is required to expand the heap is almost trivial compared to the cost of collecting and compacting a very large fragmented heap.

Using `verbose:gc`  
You can use `-verbose:gc` when running your application with no load, and again under stress, to help you set the initial and maximum heap sizes.
The `-verbose:gc` output is fully described in “Garbage Collector diagnostic data” on page 436. Turn on `-verbose:gc` and run up the application with no load. Check the heap size at this stage. This provides a rough guide to the start size of the heap (`-Xms` option) that is needed. If this value is much larger than the defaults (see “Default settings for the JVM” on page 603), 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 93.

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 values 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 81 and “Heap shrinkage” on page 82.

### Interaction of the Garbage Collector with applications

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

The Garbage Collector behaves in these ways:

1. The Garbage Collector will collect some (but not necessarily all) unreachable objects.
2. The Garbage Collector will not collect reachable objects
3. The Garbage Collector will stop all threads when it is running.
4. The Garbage Collector will start in these ways:
   a. The Garbage Collector is triggered when an allocation failure occurs, but will otherwise not run itself.
   b. The Garbage Collector will accept manual calls unless the `-Xdisableexplicitgc` parameter is specified. A manual call to the Garbage Collector (for example, through the System.gc() call) suggests that a garbage collection cycle will run. In fact, the call is interpreted as a request for full garbage collection scan unless a garbage collection cycle is already running or explicit garbage collection is disabled by specifying `-Xdisableexplicitgc`.
5. The Garbage Collector will collect garbage at its own sequence and timing, subject to item 4b.
6. The Garbage Collector accepts all command-line variables and environment variables.
7. Note these points about finalizers:
   a. They are not run in any particular sequence.
b. They are not run at any particular time.
c. They are not guaranteed to run at all.
d. They will run asynchronously to the Garbage Collector.

How to coexist with the Garbage Collector

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

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

Set up the initial conditions that you want and let the GC run. It will behave as described in "Interaction of the Garbage Collector with applications" on page 94, 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 94). This difference has significant consequences for finalizers (list item 7), which are described more fully in "Finalizers" on page 96.

Thread local heap

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

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

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

Bug reports

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

Here is an example of a regular bug report to Java service of the "Hello World" variety. A simple program allocates an object or objects, clears references to these objects, and then initiates a garbage collection cycle. The objects are not seen as collected. Typically, the objects are not collected because the application has attached a finalizer that does not run immediately.
It is clear from the way that the GC works that more than one valid reason exists for the objects not being seen as collected:

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

See list item 1 in “Interaction of the Garbage Collector with applications” on page 94. 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 94.

**Nature of finalizers:**

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

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

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

**Finalizers and garbage collection:**

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

The way finalizers work, described in list item 7 in “Interaction of the Garbage Collector with applications” on page 94, 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 94. 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 94. 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?
- What is the difference between the GC policies gencon, balanced, optavgpause, and optthruput?
- What is the default GC mode (gencon, optavgpause, or optthruput)?
- How many GC helper threads are created or “spawned”? What is their work?
- What is Mark Stack Overflow (MSO)? Why is MSO bad for performance?
- How can I prevent Mark Stack Overflow?
- When and why does the Java heap expand?
- When does the Java heap shrink?
- Does GC guarantee that it clears all the unreachable objects?
- I am getting an OutOfMemoryError. Does this mean that the Java heap is exhausted?
- When I see an OutOfMemoryError, does that mean that the Java program exits?
- In `verbose:gc` output, sometimes I see more than one GC for one allocation failure. Why?
- What do System.gc() and Runtime.gc() do?

**What are the default heap and native stack sizes?**

See “Default settings for the JVM” on page 603.

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

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

- **balanced**
  
  The balanced policy uses mark, sweep, compact and generational style garbage collection. The concurrent mark phase is disabled; concurrent
garbage collection technology is used, but not in the way that
concurrent mark is implemented for other policies. The balanced policy
uses a region-based layout for the Java heap. These regions are
individually managed to reduce the maximum pause time on large
heaps and increase the efficiency of garbage collection. The policy tries
to avoid global collections by matching object allocation and survival
rates. If you have problems with application pause times that are
caused by global garbage collections, particularly compactions, this
policy might improve application performance. If you are using large
systems that have Non-Uniform Memory Architecture (NUMA)
characteristics (x86 and POWER platforms only), the balanced policy
might further improve application throughput. For more information
about this policy, including when to use it, see “Balanced Garbage
Collection policy” on page 84.

metronome
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. The metronome policy is supported on specific
hardware and operating system configurations. For more information,
see “Using the Metronome Garbage Collector” on page 218.

optavgpause
The “optimize for pause time” (optavgpause) policy uses concurrent
mark and concurrent sweep phases. Pause times are shorter than with
optthruput, but application throughput is reduced because some
garbage collection work is taking place while the application is
running. Consider using this policy if you have a large heap size
(available on 64-bit platforms), because this policy limits the effect of
increasing heap size on the length of the garbage collection pause.
However, if your application uses many short-lived objects, the gencon
policy might produce better performance.

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

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

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

How many GC helper threads are created or “spawned”? What is their work?
The garbage collector creates \(n-1\) helper threads, where \(n\) is the number of
GC threads specified by the \(-Xgcthreads<number>\) option. For more
information, see “Garbage Collector command-line options” on page 577. 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 that are 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 by using -Xgcthreads command-line option
- Decrease the size of the Java heap by 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, for example 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 that is 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. For example, JVM through JNI.
- The heap has been fully expanded, and an excessive amount of time (95%) is being spent in the GC. This check can be disabled by 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 that is 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.

**What do System.gc() and Runtime.gc() do?**

A System.gc() call causes a global garbage collection. If the GC policy is `gencon`, both `nursery` and `tenure` areas are collected. If two explicit global garbage collections are triggered in close succession, one of these collections processes a heap compaction. You can prevent compaction during a System.gc() collection by specifying the `-Xnocompactexplicitgc` option. If there is a concurrent global garbage collection in progress when System.gc() is called, the process stops and completes as a stop-the-world global garbage collection.

There is no difference between System.gc() and Runtime.gc(). System.gc() calls Runtime.getRuntime().gc().

---

**Class loading**

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

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

- **Loading** involves obtaining the byte array representing the Java class file.
- **Verification** of a Java class file is the process of checking that the class file is structurally well-formed and then inspecting the class file contents to ensure that the code does not attempt to perform operations that are not permitted.
• **Preparation** involves the allocation and default initialization of storage space for static class fields. Preparation also creates method tables, which speed up virtual method calls, and object templates, which speed up object creation.

• **Initialization** involves the processing of the class’s class initialization method, if defined, at which time static class fields are initialized to their user-defined initial values (if specified).

Symbolic references in a Java class file, such as to classes or object fields that reference a field’s value, are resolved at run time to direct references only. This resolution might occur either:
• After preparation but before initialization
• Or, more typically, at some point following initialization, but before the first reference to that symbol.

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

**The parent-delegation model**

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

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

```
Bootstrap
| Extensions
| Application
```

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

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

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

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

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

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

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

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

Custom class loaders

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

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

1. To allow class loading from alternative repositories.
   This is the most common case, in which an application developer might want to load classes from other locations, for example, over a network connection.
2. To partition user code.
   This case is less frequently used by application developers, but widely used in servlet engines.
3. To allow the unloading of classes.
   This case is useful if the application creates large numbers of classes that are used for only a finite period. Because a class loader maintains a cache of the classes that it has loaded, these classes cannot be unloaded until the class loader itself has been dereferenced. For this reason, system and extension classes are never unloaded, but application classes can be unloaded when their class loader is.

For much more detailed information about the class loader, see [https://www.ibm.com/developerworks/java/library/j-dclp1/](https://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 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 549.

For diagnosing problems with shared classes, see “Shared classes diagnostic data” on page 463.

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 Version 7 Release 1, apart from the Oracle Solaris and HP hybrids.
- Classes are stored in a named “class cache”, which is either a memory-mapped file or an area of shared memory, allocated by the first JVM that needs to use it.
- A JVM can connect to a cache with either read-write or read-only access. Using read-only access allows greater isolation when many users are sharing the same cache.
- The class cache memory can be protected from accidental corruption using memory page protection.
- The JVM determines how many AOT methods get added to the cache and when. The amount of AOT method data is typically no more than 10% of the amount of class data cached.
- The JVM automatically removes and rebuilds a class cache if it decides that the cache is corrupted, or if the cache was created by a different level of service release.
A separate, independent cache is created for each JVM version. For example, if you are running both 31-bit and 64-bit JVMs, two caches are created. If you are running Java 6 and Java 7, two caches are created.

The file system location of the cache files can now be specified on the command line. Persistent cache files can be moved and copied around the file system. Persistent cache files can also be moved and copied between computers that use the same operating system and hardware.

Filters can be applied to Java class loaders to allow users to limit the classes being shared.

Any JVM can read from or update the cache, although a JVM can connect to only one cache at a time.

The cache persists beyond the lifetime of any JVM connected to it until it is explicitly removed. Persistent caches 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 recompile.

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 430.]

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 \texttt{-XcompilationThreads} option. For information on using this option, see “JIT and AOT command-line options” on page 573.

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
Phase 4 - global optimizations
Global optimizations work on the entire method at once. They are more “expensive”, requiring larger amounts of compilation time, but can provide a great increase in performance.

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

Phase 5 - native code generation
Native code generation processes vary, depending on the platform architecture. Generally, during this phase of the compilation, the trees of a method are translated into machine code instructions; some small optimizations are performed according to architecture characteristics.

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

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

Can I disable the JIT compiler?
Yes. The JIT compiler is turned on by default, but you can turn it off with the appropriate command-line parameter. For more information, see “Disabling the JIT or AOT compiler” on page 430.

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

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

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

Can I control the JIT compilation?
Yes. For more information, see “JIT and AOT problem determination” on page 430. In addition, advanced diagnostic settings are available to IBM engineers.
Can I dynamically control the JIT compiler?
No. You can pass options to the JIT compiler to modify the behavior, but only at JVM startup time, because the JIT compiler is started up at the same time as the JVM. However, a Java program can use the java.lang.Compiler API to enable and disable the JIT compiler at run time.

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

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

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

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

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

In a JVM without an AOT compiler or with the AOT compiler disabled, the JIT compiler selectively compiles frequently used methods into optimized native code. There is a time cost associated with compiling methods because the JIT compiler operates while the application is running. Because methods begin by being interpreted and most JIT compilations occur during startup, startup times can be increased.

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

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

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

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

The RMI implementation

Java Remote Method Invocation (RMI) provides a simple mechanism for distributed Java programming. 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.
1. 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.

DGCCClient implements the client side of the RMI distributed garbage collection system. The external interface to DGCCClient is the registerRefs() method. When a LiveRef to a remote object enters the JVM, it must be registered with the DGCCClient 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 DGCCClient periodically sends more dirty calls to renew its lease. The DGCCClient 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: https://docs.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: https://docs.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: https://docs.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 understand how the ORB works.

The topics in this chapter are:

- "CORBA" on page 113
The IBM ORB that is provided with this release is used by WebSphere Application Server. It is one of the enterprise features of the Java Standard Edition. The ORB is both a tool and a runtime component. It provides distributed computing through the CORBA Internet Inter-Orb Protocol (IIOP) communication protocol. The protocol is defined by the Object Management Group (OMG). The ORB runtime environment consists of a Java implementation of a CORBA ORB. The ORB toolkit provides APIs and tools for both the Remote Method Invocation (RMI) programming model and the Interface Definition Language (IDL) programming model.

**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.

**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):
  ```idl
  interface Sample { string message(); }
  ```

- Java RMI Interface (Sample.java):
  ```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 [https://www.omg.org](https://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:

```java
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:

```java
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 129.

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:

<table>
<thead>
<tr>
<th>CORBA</th>
<th>RMII(RJMP)</th>
<th>RMI-IIOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>idlj Sample.idl</td>
<td>rmic SampleImpl</td>
<td>rmic -iiop Sample</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>CORBA</th>
<th>RMII(RJMP)</th>
<th>RMI-IIOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample.java</td>
<td>SampleImpl_Skel.class</td>
<td>_SampleImpl_Tie.class</td>
</tr>
<tr>
<td>SampleHolder.java</td>
<td>SampleImpl_Stub.class</td>
<td>_Sample_Skel.class</td>
</tr>
<tr>
<td>SampleHelper.java</td>
<td>Sample.class (Sample.java present)</td>
<td>Sample.class (Sample.java present)</td>
</tr>
<tr>
<td>SampleOperations.java</td>
<td>SampleImpl.class (only compiled)</td>
<td>SampleImpl.class (only compiled)</td>
</tr>
<tr>
<td>_SampleStub.java</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SamplePOA.java (-fserver, -fall, -fserverTie, -fallTie)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SamplePOATie.java (-fserverTie, -fallTie)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SampleImplBase.java (-oldImplBase)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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(RJMP) 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:

```java
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:

```java
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.

```java
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
-iiopt -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 -iiopt option to enable rmic to generate the RMI-IIOP tie. If you omit
-iiopt, 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
            ........
```java
// Invoke method
System.out.println(sampleRef.message());
} catch(Exception e) {} } }

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

RMI-IIOP Client example:
public class SampleClient {
    public static void main (String [] args) {
        try {
            ORB orb = ORB.init(args, null);
            // Retrieving reference from naming service
            ......
            // Narrowing the reference to the correct class
            Sample sampleRef = (Sample)PortableRemoteObject.narrow(o, Sample.class);
            // Method Invocation
            System.out.println(sampleRef.message());
        } catch(Exception e) {} }
    }
}

Using the ORB
To use the Object Request Broker (ORB) effectively, you must understand the properties that control the behavior of the ORB.

The following table lists the property values that you can use to customize the behavior of the ORB. All property values are specified as strings. Click on the links for more information about each property.

<table>
<thead>
<tr>
<th>System property</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-Dcom.ibm.CORBA.AcceptTimeout</code> on page 527</td>
<td>Set a timeout value for an accept() call.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.AllowUserInterrupt</code> on page 528</td>
<td>Allow an interrupt to a remote method call.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.BufferSize</code> on page 528</td>
<td>Set the number of bytes to read of a GIOP message.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.ConnectionMultiplicity</code> on page 528</td>
<td>Manage the number of concurrent socket connections.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.ConnectTimeout</code> on page 529</td>
<td>Set a timeout value for opening a remote ORB connection.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.enable LocateRequest</code> on page 530</td>
<td>Configure ORB to send a LocateRequest before a Request</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.FragmentSize</code> on page 530</td>
<td>Control GIOP 1.2 fragmentation.</td>
</tr>
</tbody>
</table>
Table 4. IBM ORB system properties (continued)

<table>
<thead>
<tr>
<th>System property</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-Dcom.ibm.CORBA.FragmentTimeout</code> on page 531</td>
<td>Set a timeout value for receiving GIOP message fragments.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.GIOPAddressingDisposition</code> on page 531</td>
<td>Set the GIOP address disposition.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.InitialReferencesURL</code> on page 531</td>
<td>Configure ORB to use a server based URL instead of the bootstrap approach. Note: This property is deprecated.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.ListenerPort</code> on page 531</td>
<td>Set the server listening port.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.LocalHost</code> on page 531</td>
<td>Set the hostname of the system on which the ORB is running.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.LocateRequestTimeout</code> on page 531</td>
<td>Set a timeout value for LocateRequest messages.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.MaxOpenConnections</code> on page 532</td>
<td>Set the maximum number of in-use connections in the connection cache table.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.MinOpenConnections</code> on page 532</td>
<td>Set the minimum number of in-use connections in the connection cache table.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.NoLocalInterceptors</code> on page 532</td>
<td>Define whether local portable interceptors are used.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.ORBCharEncoding</code> on page 532</td>
<td>Specify the native encoding set.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.ORBWCharDefault</code> on page 532</td>
<td>Specify the wchar code set.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.requestRetriesCount</code> on page 532</td>
<td>Specify the number of retries when certain exceptions are received.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.requestRetriesDelay</code> on page 532</td>
<td>Specify the delay between retries.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.RequestTimeout</code> on page 533</td>
<td>Set the timeout value for a Request message.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.SendingContextRunTimeSupported</code> on page 533</td>
<td>Control the CodeBase SendingContext RunTime service</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.SendVersionIdentifier</code> on page 533</td>
<td>Determine the partner version of ORB.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.ServerSocketQueueDepth</code> on page 533</td>
<td>Set the maximum queue length for incoming connection requests.</td>
</tr>
<tr>
<td><code>-Dcom.ibm.CORBA.ShortExceptionDetails</code> on page 533</td>
<td>Control the output from a SystemException reply.</td>
</tr>
<tr>
<td><code>-Dcom.tools.rmi.c.iop.Debug</code> on page 533</td>
<td>Set the <code>rmic</code> tool to map fully qualified class names to short names.</td>
</tr>
<tr>
<td><code>-Dcom.tools.rmi.c.iop.SkipImports</code> on page 536</td>
<td>Set the <code>rmic</code> tool to generate classes with fully qualified names only.</td>
</tr>
<tr>
<td><code>-Dorg.omg.CORBA.ORBid</code> on page 548</td>
<td>Set a unique identifier for the ORB.</td>
</tr>
<tr>
<td><code>-Dorg.omg.CORBA.ORBListenEndpoints</code> on page 548</td>
<td>Identify the set of endpoints on which the ORB listens for requests.</td>
</tr>
<tr>
<td><code>-Dorg.omg.CORBA.ORBServerId</code> on page 548</td>
<td>Set the same value for all ORB instances that are running in the same server.</td>
</tr>
<tr>
<td>com.ibm.CORBA.BootstrapHost</td>
<td>This property is deprecated. It is replaced by <code>-ORBInitRef</code> and <code>-ORBDefaultInitRef</code>.</td>
</tr>
<tr>
<td>com.ibm.CORBA.BootstrapPort</td>
<td>This property is deprecated. It is replaced by <code>-ORBInitRef</code> and <code>-ORBDefaultInitRef</code>.</td>
</tr>
</tbody>
</table>

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:
<table>
<thead>
<tr>
<th>Oracle Corporation property</th>
<th>IBM property</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.sun.CORBA.ORBServerHost</td>
<td>com.ibm.CORBA.LocalHost</td>
</tr>
<tr>
<td>com.sun.CORBA.ORBServerPort</td>
<td>com.ibm.CORBA.ListenerPort</td>
</tr>
<tr>
<td>org.omg.CORBA.ORBInitialHost</td>
<td>com.ibm.CORBA.BootstrapHost</td>
</tr>
<tr>
<td>org.omg.CORBA.ORBInitialPort</td>
<td>com.ibm.CORBA.BootstrapPort</td>
</tr>
<tr>
<td>org.omg.CORBA.ORBInitialServices</td>
<td>com.ibm.CORBA.InitialReferencesURL</td>
</tr>
</tbody>
</table>

**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 115.

**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 122
- "Obtaining the remote object” on page 123
- "Remote method invocation” on page 124

**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 process that is run by the init() method, the ORB is fully functional and ready to support the client application.

Obtaining the remote object:

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

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

IOR:000000000000001d524d493a5......

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

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

49424d0a 00000008 00000000 1400 0005

In the example:

• The first three bytes are the ASCII code for IBM
• The next byte is 0x0A, which specifies that the following bytes provide information about the partner version.
• The next 4 bytes encode the length of the remaining data. In this example, the remaining data is 8 bytes long.
• The next 4 null bytes are reserved for future use.
• The next 2 bytes are for the Partner Version Major field. In this example, the value is 0x1400, which means that release 1.4.0 of the ORB is being used.
• The final 2 bytes in this example have the value 0x0005 and represent the Minor field. This field is used to distinguish maintenance updates within the same release. The updates 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 293 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.

Handling request retries:

When a CORBA request results in a response of type
org.omg.CORBA.SystemException, the IBM ORB retries a request under certain
conditions. You can control the number of retries and the time delay between
retries by using system properties.

When a CORBA request results in a response of type
org.omg.CORBA.SystemException, the IBM ORB retries a request when the
following two conditions are met:
1. The org.omg.CORBA.SystemException is one of the following types:
   • org.omg.CORBA.COMM_FAILURE
   • org.omg.CORBA.TRANSENT
   • org.omg.CORBA.NO_RESOURCE
2. The completion status of the org.omg.CORBA.SystemException is
   org.omg.CORBA.CompletionStatus.COMPLETED_NO.

The following properties govern ORB retry behavior:

com.ibm.CORBA.requestRetriesCount
   This property governs the number of retries that are attempted.

com.ibm.CORBA.requestRetriesDelay
   This property determines the time delay in milliseconds (ms) between
   retries.

For more information about these properties, see
"-Dcom.ibm.CORBA.requestRetriesCount" on page 532 and
"-Dcom.ibm.CORBA.requestRetriesDelay" on page 532.

The overall time delay before the client receives an
org.omg.CORBA.SystemException response if all retry attempts fail, is based on the
property values for these two properties. Note: IBM ORB does not retry a request
when a completion status that contains COMPLETED_MAYBE or
COMPLETED_YES is returned.

The Interoperable Object References (IOR) also play a role in the retry logic. The
ORB deals with two different IORs when CORBA requests are sent, which are the
“indirect” (or “initial”) IOR and “direct” (or “forwarded”) IOR. The “initial” IOR is
the bootstrap IOR with which a client starts the initial request, which is essentially
an IOR used to contact the NameService on the bootstrap port. Subsequent
requests are made by using the “forwarded” IOR, which is a “direct” IOR to a
specific server’s ORB port. If an org.omg.CORBA.SystemException exception is
thrown of one of the specified types, the ORB falls back to the “initial” IOR and
sends the retry request by using the “initial” IOR. This behavior allows the ORB to
obtain a new “direct” IOR to a server to attempt to complete the request. The
IOR-related logic is separate from the property settings that are made by using
com.ibm.CORBA.requestRetriesCount.

Important: The retry behavior of the two IOR types is not configurable, is internal
to IBM ORB, and might be changed without any prior notification.
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 128

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”
- “Interoperable Naming Service (INS)” on page 131

**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 v2.3, 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.

The IBM ORB supports both the POA specification and the previous proprietary basic object adapter. 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.

**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:iop:<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:iop: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](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/](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 133
- "Copying and pinning" on page 138
- "Handling exceptions" on page 139
- "Synchronization" on page 140
- "Debugging the JNI" on page 140
- "JNI checklist" on page 142

**Overview of JNI**

From the viewpoint of a JVM, there are two types of code: "Java" and "native". The Java Native Interface (JNI) establishes a well-defined and platform-independent interface between the two.

Native code can be used together with Java in two distinct ways: as "native methods" in a running JVM and as the code that creates a JVM using the "Invocation API". This section describes the difference.

**Native methods**

Java native methods are declared in Java, implemented in another language (such as C or C++), and loaded by the JVM as necessary. To use native methods, you must:

1. **Declare** the native method in your Java code.
   
   When the javac compiler encounters a native method declaration in Java source code, it records the name and parameters for the method. Because the Java source code contains no implementation, the compiler marks the method as "native". The JVM can then resolve the method correctly when it is called.

2. **Implement** the native method.
Native methods are implemented as external entry points in a loadable binary library. The contents of a native library are platform-specific. The JNI provides a way for the JVM to use any native methods in a platform-independent way. The JVM performs calls to native methods. When the JVM is in a native method, JNI provides a way to "call back" to the JVM.

3. **Load** the native method code for the VM to use.

As well as declaring the native method, you must find and load the native library that contains the method at run time.

Two Java interfaces load native libraries:

- `java.lang.System.load()`
- `java.lang.System.loadLibrary()`

Typically, a class that declares native methods loads the native library in its static initializer.

**Invocation API**

Creating a JVM involves native code. The aspect of the JNI used for this purpose is called the JNI Invocation API. To use the Invocation API, you bind to an implementation-specific shared library, either statically or dynamically, and call the `JNI_*` functions it exports.

**The JNI specification and implementation**

The JNI specification is vague on selected implementation details. It provides a reusable framework for simple and extensible C and C++ native interfaces. The JNI model is also the basis for the JVMTI specification.

The Oracle Corporation trademark specification and the Java Compatibility Kit (JCK) ensure compliance to the specification but not to the implementation. Native code must conform to the specification and not to the implementation. Code written against unspecified behavior is prone to portability and forward compatibility problems.

**The JNI and the Garbage Collector**

This description explains how the JNI implementation ensures that objects can be reached by the Garbage Collector (GC).

For general information about the IBM GC, see the section on memory management.

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 the section on detailed description of global garbage collection 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 139.

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:

```java
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 );
    }
}
```
Although new local references overwrite the myObj and myClazz variables inside the loop, every local reference is kept in the root set. These references must be explicitly removed by the DeleteLocalRef call. Without the DeleteLocalRef calls, the local references are leaked until the thread returned to Java or detached from the JVM.

**JNI weak global references**

Weak global references are a special type of global reference. They can be used in any thread and can be used between native function calls, but do not act as GC roots. The GC disposes of an object that is referred to by a weak global reference at any time if the object does not have a strong reference elsewhere.

You must use weak global references with caution. If the object referred to by a weak global reference is garbage collected, the reference becomes a null reference. A null reference can only safely be used with a subset of JNI functions. To test if a weak global reference has been collected, use the IsSameObject JNI function to compare the weak global reference to the null value.

It is not safe to call most JNI functions with a weak global reference, even if you have tested that the reference is not null, because the weak global reference could become a null reference after it has been tested or even during the JNI function. Instead, a weak global reference should always be promoted to a strong reference before it is used. You can promote a weak global reference using the NewLocalRef or NewGlobalRef JNI functions.

Weak global references use memory and must be freed with the DeleteWeakGlobalRef JNI function when it is no longer needed. Failure to free weak global references causes a slow memory leak, eventually leading to out-of-memory exceptions.

For information and warnings about the use of JNI global weak references, see the JNI specification.

**JNI reference management**

There are a set of platform-independent rules for JNI reference management

These rules are:
1. JNI references are valid only in threads attached to a JVM.
2. A valid JNI local reference in native code must be obtained:
   a. As a parameter to the native code
   b. As the return value from calling a JNI function
3. A valid JNI global reference must be obtained from another valid JNI reference (global or local) by calling NewGlobalRef or NewWeakGlobalRef.
4. The null value reference is always valid, and can be used in place of any JNI reference (global or local).
5. JNI local references are valid only in the thread that creates them and remain valid only while their creating frame remains on the stack.
Note:

1. Overwriting a local or global reference in native storage with a null value does not remove the reference from the root set. Use the appropriate Delete*Ref JNI function to remove references from root sets.

2. Many JNI functions (such as FindClass and NewObject) return a null value if there is an exception pending. Comparing the returned value to the null value for these calls is semantically equivalent to calling the JNI ExceptionCheck function. See the JNI specification for more details.

3. A JNI local reference must never be used after its creating frame returns, regardless of the circumstances. It is dangerous to store a JNI local reference in any process static storage.

JNI transitions

To understand JNI local reference management and the GC, you must understand the context of a running thread attached to the JVM. Every thread has a runtime stack that includes a frame for each method call. From a GC perspective, every stack establishes a thread-specific "root set" including the union of all JNI local references in the stack.

Each method call in a running VM adds (pushes) a frame onto the stack, just as every return removes (pops) a frame. Each call point in a running stack can be characterized as one of the following types:

- Java to Java (J2J)
- Native to Native (N2N)
- Java to Native (J2N)
- Native to Java (N2J)

You can only perform an N2J transition in a thread that meets the following conditions:

- The process containing the thread must contain a JVM started using the JNI Invocation API.
- The thread must be "attached" to the JVM.
• The thread must pass at least one valid local or global object reference to JNI.

J2J and N2N transitions:

Because object references do not change form as part of J2J or N2N transitions, J2J and N2N transitions do not affect JNI local reference management.

Any section of N2N code that obtains many local references without promptly returning to Java can needlessly stress the local reference capacity of a thread. This problem can be avoided if local references are managed explicitly by the native method programmer.

N2J transitions:

For native code to call Java code (N2J) in the current thread, the thread must first be attached to the JVM in the current process.

Every N2J call that passes object references must have obtained them using JNI, therefore they are either valid local or global JNI refs. Any object references returned from the call are JNI local references.

J2N calls:

The JVM must ensure that objects passed as parameters from Java to the native method and any new objects created by the native code remain reachable by the GC. To handle the GC requirements, the JVM allocates a small region of specialized storage called a local reference root set.

A local reference root set is created when:
• A thread is first attached to the JVM (the outermost root set of the thread).
• Each J2N transition occurs.

The JVM initializes the root set created for a J2N transition with:
• A local reference to the caller’s object or class.
• A local reference to each object passed as a parameter to the native method.

New local references created in native code are added to this J2N root set, unless you create a new local frame using the PushLocalFrame JNI function.

The default root set is large enough to contain 16 local references per J2N transition. The -Xcheck:jni command-line option causes the JVM to monitor JNI usage. When -Xcheck:jni is used, the JVM writes a warning message when more than 16 local references are required at run time. If you receive this warning message, use one of the following JNI functions to manage local references more explicitly:
• NewLocalRef
• DeleteLocalRef
• PushLocalFrame
• PopLocalFrame
• EnsureLocalCapacity

J2N returns:

When native code returns to Java, the associated JNI local reference "root set", created by the J2N call, is released.
If the JNI local reference was the only reference to an object, the object is no longer reachable and can be considered for garbage collection. Garbage collection is triggered automatically by this condition, which simplifies memory management for the JNI programmer.

**Copying and pinning**

The GC might, at any time, decide it needs to compact the garbage-collected heap. Compaction involves physically moving objects from one address to another. These objects might be referred to by a JNI local or global reference. To allow compaction to occur safely, JNI references are not direct pointers to the heap. At least one level of indirection isolates the native code from object movement.

If a native method needs to obtain direct addressability to the inside of an object, the situation is more complicated. The requirement to directly address, or pin, the heap is typical where there is a need for fast, shared access to large primitive arrays. An example might include a screen buffer. In these cases a JNI critical section can be used, which imposes additional requirements on the programmer, as specified in the JNI description for these functions. See the JNI specification for details.

- GetPrimitiveArrayCritical returns the direct heap address of a Java array, disabling garbage collection until the corresponding ReleasePrimitiveArrayCritical is called.
- GetStringCritical returns the direct heap address of a java.lang.String instance, disabling garbage collection until ReleaseStringCritical is called.

All other Get<PrimitiveType>ArrayElements interfaces return a copy that is unaffected by compaction.

When using the Balanced Garbage Collection Policy, the *Critical forms of the calls might not return a direct pointer into the heap, which is reflected in the isCopy flag. This behavior is due to an internal representation of larger arrays, where data might not be sequential. Typically, an array with storage that is less than 1/1000th of the heap, is returned as a direct pointer.

**Using the isCopy flag**

The JNI Get<Type> functions specify a pass-by-reference output parameter (jboolean *isCopy) that allows the caller to determine whether a given JNI call is returning the address of a copy or the address of the pinned object in the heap.

The Get<Type> and Release<Type> functions come in pairs:

- GetStringChars and ReleaseStringChars
- GetStringCritical and ReleaseStringCritical
- GetStringUTFChars and ReleaseStringUTFChars
- Get<PrimitiveType>ArrayElements and Release<PrimitiveType>ArrayElements
- GetPrimitiveArrayCritical and ReleasePrimitiveArrayCritical

If you pass a non-null address as the isCopy parameter, the JNI function sets the jboolean value at that address to JNI_TRUE if the address returned is the address of a copy of the array elements and JNI_FALSE if the address points directly into the pinned object in the heap.

Except for the critical functions, the IBM VM 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" on page 140.

**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:valist 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:

<table>
<thead>
<tr>
<th>Remember</th>
<th>Outcome of nonadherence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local references cannot be saved in global variables.</td>
<td>Random crashes (depending on what you pick up in the overwritten object space) happen at random intervals.</td>
</tr>
<tr>
<td>Ensure that every global reference created has a path that deletes that global reference.</td>
<td>Memory leak. It might throw a native exception if the global reference storage overflows. It can be difficult to isolate.</td>
</tr>
<tr>
<td>Always check for exceptions (or return codes) on return from a JNI function. Always handle a deferred exception immediately you detect it.</td>
<td>Unexplained exceptions or undefined behavior. Might crash the JVM.</td>
</tr>
<tr>
<td>Ensure that array and string elements are always freed.</td>
<td>A small memory leak. It might fragment the heap and cause other problems to occur first.</td>
</tr>
<tr>
<td>Ensure that you use the isCopy and mode flags correctly. See &quot;A generic way to use the isCopy and mode flags&quot; on page 139.</td>
<td>Memory leaks, heap fragmentation, or both.</td>
</tr>
<tr>
<td>When you update a Java object in native code, ensure synchronization of access.</td>
<td>Memory corruption.</td>
</tr>
</tbody>
</table>
Chapter 3. Planning

Information that you should know when planning to use or migrate the product, such as supported environments and version compatibility.

Migrating from earlier releases of the IBM SDK, Java Technology Edition

Important information to consider before upgrading from earlier releases of the IBM SDK, Java Technology Edition.

This release contains many new features and functions, which might require careful planning. For an overview, read the “What’s new” section.

If you are migrating from IBM SDK, Java Technology Edition Version 7, note that:

• During startup, the JVM processes options in a different order from earlier releases. Options that are created automatically by the JVM, such as search paths and version information, are constructed first and take the lowest precedence. For more information about construction order, see “Specifying command-line options” on page 525.

• The -Xzero option is deprecated in this release and will be removed from future releases.

• The default behavior of the method java.nio.ByteBuffer.allocateDirect(int) changed in Version 7. Direct byte buffers are no longer allocated on a page boundary. Any applications that rely on the earlier undocumented alignment, which occurred in Version 6, can revert to the previous behavior with -XX:+PageAlignDirectMemory. For more information, see “JVM -XX:command-line options” on page 569.

• The JVM file naming convention for the shared cache has changed.

• The Byte Code Instrumentation (BCI) implementation has changed, which can affect performance and behavior when you are migrating from earlier releases. Note the following points:
  – Class data is no longer stored and must be created. This change reduces the footprint requirements.
  – When shared classes are enabled, the throughput of calling the ClassFileLoadHook on initial class load is decreased since the introduction of -Xshareclasses:enableBCI. This throughput reduction occurs because the shared cache no longer stores a copy of the class data.
  – The class data that is provided to the ClassFileLoadHook on retransformation of classes is no longer an exact copy of the original class file, but is equivalent. This change might cause agents that expected earlier release behavior to fail. Agents that work with the Oracle JDK should not experience problems. The class data now has annotations with the CLASS retention policy removed, which was not the case in previous releases.
  – The option -Xshareclasses:enableBCI is now enabled by default.
  – If your application uses the -Xshareclasses:modified= option, you must also set -Xshareclasses:disableBCI to retain the same behavior.
  – Raw class data is no longer stored in the shared classes cache. If you previously increased the size of your shared classes cache to accommodate raw class data, the size can now be reduced.
By default, the JVM forces synchronization on a class loader that is not a parallel capable class loader during class loading. This action occurs even if the loadClass() method for that classloader is not synchronized. To revert to the earlier behavior of Version 7, use the `-XX:VMLockClassLoader` option, described in "JVM -XX: command-line options" on page 569.

Changes are made to the content of Javadumps in this release. For more information, see "Interpreting a Javadump" on page 342.

The file format of portable heap dumps (PHD) is changed to PHD Version 6. For more information, see "Portable Heap Dump (PHD) file format" on page 367.

The Diagnostics Collector utility is removed from this release onwards. Instead, use the IBM Support Assistant Data Collector to collect diagnostic data, and to send that data to IBM, if required. For more information, see "IBM Support Assistant Data Collector" on page 436.

If you are migrating from a release of IBM SDK, Java Technology Edition Version 7 before service refresh 7 fix pack 1, the currency code symbol for Zambia is now corrected to the value “ZMW”.

On 64-bit platforms, the total code cache space has a limit of 256 MB. If this limit is not sufficient, you can increase it by using the "-Xcodecachetotal" option on page 574.

Unsigned or self-signed applets no longer run by default because the security settings are changed to high. For more information about this change, see "Default behaviour change when running unsigned or self-signed applets" on page 609.

The default value of the com.ibm.CORBA.FragmentSize property was changed in service refresh 1 from 1024 bytes to 4096 bytes. For more information about this property, see "Using the ORB" on page 119.

For Linux on IBM Z® platforms, if you are migrating your hardware to IBM z13 you must install IBM SDK, Java Technology Edition Version 7 Release 1 Service Refresh 2 Fix Pack 10 or later to avoid a performance degradation.

As announced in the Oracle Java SE Support roadmap, support for Java plug-in and Web Start technology in Java SE 7 ends in July 2016. Accordingly, support is also removed from IBM SDK, Java Technology Edition, Version 7 Release 1. For continued support, move to the latest release of the IBM SDK for all applications launched or deployed through web browsers.

If you are migrating from Version 6, you should consider the following information:

- The default garbage collection policy in force is now the Generational Concurrent garbage collector. For an overview, see "Generational Concurrent Garbage Collector" on page 83.
- The JIT compiler can use more than one thread to convert method bytecodes into native code, dynamically. If the default number of threads that are chosen by the JVM is not optimum for your environment, you can configure the number of threads by setting a system property. For more information, see "How the JIT compiler optimizes code" on page 106.
- Shared class caches are now persistent by default on the AIX operating system. For more information, see "-Xshareclasses" on page 560.
- Compressed references are now the default on all platforms except z/OS when the value of the `-Xmx` option is less than or equal to 25 GB. For more information about the use of compressed references, see "Compressed references" on page 73.
• Verbose garbage collection logging is redesigned. See “Verbose garbage collection logging” on page 437.

• The default value for -Xjni:arrayCacheMax is increased from 8096 bytes to 128 KB. Because more memory is used, this change might lead to an out of memory error. For more information, see “-Xjni” on page 554.

• From Service Refresh 4: the Oracle implementation of the java.util.* package is used, including all classes within the package. Earlier releases used customized versions of the Apache Harmony class libraries. This change establishes a common implementation point for the java.util.* package, enabling consistent performance and behavior characteristics across Java technology implementations.

• From Service Refresh 6 fix pack 1: Unsigned or self-signed applets no longer run by default because the security settings are changed to high.

• If you are migrating from a release of IBM SDK, Java Technology Edition Version 6 before service refresh 16 fix pack 1, the currency code symbol for Zambia is now corrected to the value “ZMW”.

• Support is no longer available for the following hardware platforms:
  – IBM POWER 3

• The default behavior of the method java.nio.ByteBuffer.allocateDirect(int) changed in Version 7. Direct byte buffers are no longer allocated on a page boundary. Any applications that rely on the earlier undocumented alignment, which occurred in Version 6, can revert to the previous behavior with -XX:+PageAlignDirectMemory. For more information, see “JVM -XX:command-line options” on page 569.

• As announced in the Oracle Java SE Support roadmap, support for Java plug-in and Web Start technology in Java SE 7 ends in July 2016. Accordingly, support is also removed from IBM SDK, Java Technology Edition, Version 7 Release 1. For continued support, move to the latest release of the IBM SDK for all applications launched or deployed through web browsers.

• For additional industry compatibility information, see “Version compatibility.”

• For additional deprecated API information, see Oracle’s Java 7 Deprecated API List: https://docs.oracle.com/javase/7/docs/api/deprecated-list.html

If you are migrating from a release before Version 6, read the topic entitled “Migrating from earlier releases” in the appropriate platform user guide for Version 6. These user guides can be found at the following URL:


**Version compatibility**

In general, any applet or application that ran with a previous version of the SDK should run correctly with this release. Classes that are compiled with this release are not guaranteed to work on previous releases.

For information about compatibility issues between releases, see the Oracle website at:

http://www.oracle.com/technetwork/java/javase/compatibility-417013.html
Supported environments

This release is supported on certain hardware platforms and operating systems, and is tested on specific virtualization environments.

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

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

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

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

<table>
<thead>
<tr>
<th>Operating system</th>
<th>32-bit release</th>
<th>64-bit release</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX 6.1 TL7</td>
<td>GA</td>
<td>GA</td>
</tr>
<tr>
<td>AIX 7.1 TL1</td>
<td>GA</td>
<td>GA</td>
</tr>
<tr>
<td>AIX 7.2</td>
<td>SR3 FP20</td>
<td>SR3 FP20</td>
</tr>
</tbody>
</table>

**Table 5. Supported AIX environments**

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

Specialized hardware features

The following hardware features have additional prerequisites to those listed in this topic:

- Hardware compression acceleration: "Enabling hardware compression acceleration on Power Systems" on page 246

Virtualization software

For information about the virtualization software tested, see “Support for virtualization software” on page 609.

Additional information for AIX

Important information for this release on AIX platforms and architectures.

AIX APARs required for this release

To avoid problems when using Java, ensure that you have any prerequisite AIX APARs installed. For further information about the APARs needed for an AIX level, see [https://www.ibm.com/support/docview.wss?uid=swg21605167](https://www.ibm.com/support/docview.wss?uid=swg21605167)
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 this release on 64-bit AIX, 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](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](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 the operation of this release. If you encounter this effect, see APAR [https://www-304.ibm.com/support/docview.wss?uid=isg1IV05072](https://www-304.ibm.com/support/docview.wss?uid=isg1IV05072)

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 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. For more information, see "Installing from an InstallAnywhere package" on page 150. Packages are provided for both the Java runtime environment 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:
- Java71.sdk (license, base SDK, Web Start, and dt.jar)
- Java71_64.sdk (license, base SDK and dt.jar)

These packages are optional:
- Java71.samples (demos)
- Java71.jclsource (src.jar)
- Java71.sampsource (src.jar)
- Java71.msg.$LANG (Localized messages)
- Java71_64.samples (demos)
- Java71_64.jclsource (src.jar)
- Java71_64.sampsource (src.jar)
- Java71_64.msg.$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

Installp packages are installed by the AIX `installp` command. For more information about using this command, see the AIX product documentation. For example, if you are using AIX 7.1, see [https://www.ibm.com/support/knowledgecenter/ssw_aix_71/com.ibm.aix.cmds3/installp.html](https://www.ibm.com/support/knowledgecenter/ssw_aix_71/com.ibm.aix.cmds3/installp.html).
The SDK is installed in the directory:

/usr/java71[/64]/

The following user-configurable files are installed to /etc/java71[/64] to support a configuration where the files are not shared:

- jre/lib/jaxp.properties
- jre/lib/logging.properties
- 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/java71[/64] pointing to the files in /etc/java71[/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 152.

**Completing an attended installation**

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

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. To read the installation instructions in another language, select a language from the list that is 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. To proceed with the installation, you must accept the terms of the license agreement. To accept the terms, read to the end of the license text by using the scroll bar. Select the radio button, then click OK.

   Note: To read the license agreement in your chosen language, you might need to change your system locale.
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. The unattended process uses a response file to complete installations without any user interaction.

Before you begin

Check the conditions that are documented in “Completing an attended installation” on page 150.

About this task

Before you use the unattended installation process, you must accept the terms of the license agreement. You can do this by running an attended installation to generate a new response file that sets a specific value, or by reading the license agreement and manually updating an existing response file. More information is provided in the first step.

Procedure
1. To create a new response file, complete 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:

```ini
INSTALLER_UI=silent
USER_INSTALL_DIR=/my_directory
LICENSE_ACCEPTED=TRUE
```

In this example, /my_directory is the target installation directory that you chose for the IBM SDK or Java runtime environment.

**Note:** The value LICENSE_ACCEPTED=TRUE is added when you create the response file by running an attended installation and accepting the license agreement. If you edit an existing response file, you must read the license agreement and include this line to confirm your license acceptance, or the installation fails.

2. Optional: If required, edit the response file to change options.

   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.
      
      ```bash
      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.
      
      ```bash
      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:

   ```bash
   ./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 runtime environment installation, if it was created. For example `/usr/java71[_64]/.`
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/java71_[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 .' Java71_64.sdk

    Remove the SDK:
    installp_r -u -R /<Install Path>/ Java71_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 enter the following commands:

```
unset LIBPATH
unset CLASSPATH
unset JAVA_COMPILER
export PATH=/usr/java71[_64]/jre/bin:/usr/java71[_64]/bin:$PATH
```

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_27-20130715_03)
IBM J9 VM (build 2.7, JRE 1.7.0 IBM J9 2.7 AIX ppc-32 jvmap3270-20130711_156087
(JIT enabled, AOT enabled)
J9VM - R27_head_20130711_0404_B156087
JIT - tr.r13.java_20130709_41534
GC - R27_head_20130711_0404_B156087
J9CL - 20130711_156087)
JCL - 20130704_01 based on Oracle 7u25-b12
```

- For AIX 64-bit:

```
java version "1.7.0"
Java(TM) SE Runtime Environment (build pap6470_27-20130715_03)
IBM J9 VM (build 2.7, JRE 1.7.0 IBM J9 2.7 AIX ppc64-64 jvmap3270-20130711_156087
(JIT enabled, AOT enabled)
J9VM - R27_head_20130711_0404_B156087
JIT - tr.r13.java_20130709_41534
GC - R27_head_20130711_0404_B156087
J9CL - 20130711_156087)
JCL - 20130704_01 based on Oracle 7u25-b12
```

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 command at a command prompt:

```
echo $PATH
```

To add the Java launchers to your path:

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

   ```
echo 'export PATH=/usr/java71[64]/bin:/usr/java71[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.
You can apply recent changes to Daylight Saving Time by 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 1 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 IBM developer kits and Runtime Environments contain up to date DST information, IBM incorporates the latest Olson time zone level into every updated release. To find out which Olson time zone level is included for a particular SDK or Runtime level, see [https://www.ibm.com/developerworks/java/jdk/dst/olson_table.html](https://www.ibm.com/developerworks/java/jdk/dst/olson_table.html).

If a DST change has been introduced since the last IBM update of the SDK or Runtime Environment, 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 SDK or Runtime level. JTZU is available from IBM developerWorks at the following link: [https://www.ibm.com/developerworks/java/jdk/dst/jtzu.html](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 contains many tools and libraries required for Java software development.

See "IBM Software Developers Kit (SDK)" on page 2 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 158. 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 162 section for more details about what is supported by XL XP-J 1.1.

XL TXE-J 1.0
For Version 5.0, the IBM SDK for Java included the XSLT4J compiler and interpreter. The XSLT4J interpreter was used by default.

For Version 6 and later, the IBM SDK for Java includes XL TXE-J. XL TXE-J includes the XSLT4J 2.7.8 interpreter and a new XSLT compiler. The new compiler is used by default. The XSLT4J compiler is no longer included with the IBM SDK for Java. See "Migrating to the XL-TXE-J" on page 159 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/java71[_64]/jre/lib/stax.properties.
   - For other factories. The value of the service provider in the file /etc/java71[_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`

`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
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 'https://www.ibm.com/xmlns/prod/xltxe-j/
'split-limit' transformer factory attribute. You can use the compiled classes, but you might get better performance by controlling the split limit manually.

- If the compiler does not split the method successfully, you receive one of the following exceptions:
  
  com.ibm.xtq.bcel.generic.ClassGenException: Branch target offset too large for short or bytecode array size > 65535 at offset=###
  
  Try setting the split limit manually, or decreasing the split limit.

To set the split limit, use the -SPLITLIMIT option when using the Process or Compile commands, or the https://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>
<thead>
<tr>
<th>XSL4J compiler attribute</th>
<th>XL TXE-J compiler attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>translet-name</td>
<td><a href="https://www.ibm.com/xmlns/prod/xltxe-j/translet-name">https://www.ibm.com/xmlns/prod/xltxe-j/translet-name</a></td>
</tr>
<tr>
<td>package-name</td>
<td><a href="https://www.ibm.com/xmlns/prod/xltxe-j/package-name">https://www.ibm.com/xmlns/prod/xltxe-j/package-name</a></td>
</tr>
<tr>
<td>jar-name</td>
<td><a href="https://www.ibm.com/xmlns/prod/xltxe-j/jar-name">https://www.ibm.com/xmlns/prod/xltxe-j/jar-name</a></td>
</tr>
<tr>
<td>debug</td>
<td><a href="https://www.ibm.com/xmlns/prod/xltxe-j/debug">https://www.ibm.com/xmlns/prod/xltxe-j/debug</a></td>
</tr>
<tr>
<td>indent-number</td>
<td><a href="https://www.ibm.com/xmlns/prod/xltxe-j/indent-number">https://www.ibm.com/xmlns/prod/xltxe-j/indent-number</a></td>
</tr>
<tr>
<td>enable-inlining</td>
<td>Obsolete in new compiler</td>
</tr>
</tbody>
</table>

6. Optional: For best performance, ensure that you are not recompiling XSLT transformations that can be reused. Use one of the following methods to reuse compiled transformations:

- If your stylesheet does not change at run time, compile the stylesheet as part of your build process and put the compiled classes on your classpath. Use the org.apache.xalan.xsltc.cmdline.Compile command to compile the stylesheet and set the https://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 https://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
transformer factory attribute to set the directory used to store compiled stylesheets. By default, compiled stylesheets are stored in the same directory as the stylesheet.

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

- If your application uses each stylesheet just once or a very small number of times, or you are unable to make any of the other changes listed in this step, you might want to continue to use the XSLT4J interpreter by setting the javax.xml.transform.TransformerFactory service provider to org.apache.xalan.processor.TransformerFactoryImpl.

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

If your application takes untrusted XML, XSD or XSL files as input, you can enforce specific limits during JAXP processing to protect your application from malformed data. If you specify limits, you must override the default XML parser configuration with a custom configuration.

About this task

To protect your application from malformed data, you can enforce specific limits during JAXP processing. These limits can be set in your jaxp.properties file, or by specifying various system properties on the command line. However, for these limits to take effect you must also override the default XML parser configuration with a custom configuration that allows these secure processing limits.

Procedure

1. Select the limits that you want to set for your application.
   - To limit the number of entity expansions in an XML document, see “-Djdk.xml.entityExpansionLimit” on page 542.
   - To limit the maximum size of a general entity, see “-Djdk.xml.maxGeneralEntitySizeLimit” on page 542.
   - To limit the maximum size of a parameter entity, see “-Djdk.xml.maxParameterEntitySizeLimit” on page 544.
   - To limit the length of XML names in XML documents, see “-Djdk.xml.maxXMLNameLimit” on page 545.
   - To limit the total size of all entities that include general and parameter entities, see “-Djdk.xml.totalEntitySizeLimit” on page 546.
   - To define the maximum number of content model nodes that can be created in a grammar, see “-Djdk.xml.maxOccur” on page 543.
   - To control whether external entities are resolved in an XML document, see “-Djdk.xml.resolveExternalEntities” on page 545.
2. To override the default XML parser configuration, set the custom configuration by specifying the following system property on the command line: -Dorg.apache.xerces.xni.parser.XMLParserConfiguration=configuration_file, where configuration_file is org.apache.xerces.parsers.SecureProcessingConfiguration. For more information about the full override mechanism, see http://xerces.apache.org/xerces2-j/faq-xni.html#faq-2.
XML reference information

The XL XP-J and XL TXE-J XML libraries are new for Version 6 of the SDK. This reference information describes the features supported by these libraries.

XL XP-J reference information

XL XP-J 1.1 is a high-performance non-validating parser that provides support for StAX 1.0 (JSR 173). StAX is a bidirectional API for pull-parsing and streaming serialization of XML 1.0 and XML 1.1 documents.

Unsupported features

The following optional StAX features are not supported by XL XP-J:

- DTD validation when using an XMLStreamReader or XMLEventReader. The XL XP-J parser is non-validating.
- When using an XMLStreamReader to read from a character stream (java.io.Reader), the Location.getCharaterOffset() method always returns -1. The Location.getCharaterOffset() returns the byte offset of a Location when using an XMLStreamReader to read from a byte stream (java.io.InputStream).

XMLInputFactory reference

The javax.xml.stream.XMLInputFactory implementation supports the following properties, as described in the XMLInputFactory Javadoc information: https://docs.oracle.com/javase/7/docs/api/javax/xml/stream/XMLInputFactory.html.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Supported?</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.xml.stream.isValidating</td>
<td>No. The XL XP-J scanner does not support validation.</td>
</tr>
<tr>
<td>javax.xml.stream.isNamespaceAware</td>
<td>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.</td>
</tr>
<tr>
<td>javax.xml.stream.isCoalescing</td>
<td>Yes</td>
</tr>
<tr>
<td>javax.xml.stream.isReplacingEntityReferences</td>
<td>Yes. For XMLStreamReaders created from DOMSources, if entities have already been replaced in the DOM tree, setting this parameter has no effect.</td>
</tr>
<tr>
<td>javax.xml.stream.isSupportingExternalEntities</td>
<td>Yes</td>
</tr>
<tr>
<td>javax.xml.stream.supportDTD</td>
<td>True is always supported. Setting the value to false works only if the com.ibm.xml.xlxp.support.dtd.comapt.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.</td>
</tr>
<tr>
<td>javax.xml.stream.reporter</td>
<td>Yes</td>
</tr>
<tr>
<td>javax.xml.streamresolver</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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: [https://
docs.oracle.com/javase/7/docs/api/javax/xml/stream/XMLStreamReader.html](https://docs.oracle.com/javase/7/docs/api/javax/xml/stream/XMLStreamReader.html)

<table>
<thead>
<tr>
<th>Property name</th>
<th>Supported?</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.xml.stream.entities</td>
<td>Yes</td>
</tr>
<tr>
<td>javax.xml.stream.notations</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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: [https://
docs.oracle.com/javase/7/docs/api/javax/xml/stream/XMLOutputFactory.html](https://docs.oracle.com/javase/7/docs/api/javax/xml/stream/XMLOutputFactory.html)

<table>
<thead>
<tr>
<th>Property name</th>
<th>Supported?</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.xml.stream.isRepairingNamespaces</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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: [https://
docs.oracle.com/javase/7/docs/api/javax/xml/stream/XMLStreamWriter.html](https://docs.oracle.com/javase/7/docs/api/javax/xml/stream/XMLStreamWriter.html)

<table>
<thead>
<tr>
<th>Property name</th>
<th>Supported?</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.xml.stream.isRepairingNamespaces</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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 7. Comparison of the features in the XSLT4J interpreter, the XSLT4J compiler, and the XL TXE-J compiler.*

<table>
<thead>
<tr>
<th>Feature</th>
<th>XSLT4J interpreter (included)</th>
<th>XSLT4J compiler (not included)</th>
<th>XL TXE-J compiler (included)</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://javax.xml.transform.stream.StreamSource/feature">http://javax.xml.transform.stream.StreamSource/feature</a> feature</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><a href="http://javax.xml.transform.stream.StreamResult/feature">http://javax.xml.transform.stream.StreamResult/feature</a> feature</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><a href="http://javax.xml.transform.dom.DOMSource/feature">http://javax.xml.transform.dom.DOMSource/feature</a> feature</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><a href="http://javax.xml.transform.dom.DOMResult/feature">http://javax.xml.transform.dom.DOMResult/feature</a> feature</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><a href="http://javax.xml.transform.sax.SAXSource/feature">http://javax.xml.transform.sax.SAXSource/feature</a> feature</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><a href="http://javax.xml.transform.sax.SAXResult/feature">http://javax.xml.transform.sax.SAXResult/feature</a> feature</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><a href="http://javax.xml.transform.stax.StAXSource/feature">http://javax.xml.transform.stax.StAXSource/feature</a> feature</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><a href="http://javax.xml.transform.stax.StAXResult/feature">http://javax.xml.transform.stax.StAXResult/feature</a> feature</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><a href="http://javax.xml.transform.sax.SAXTransformerFactory/feature">http://javax.xml.transform.sax.SAXTransformerFactory/feature</a> feature</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><a href="http://javax.xml.transform.sax.SAXTransformerFactory/feature/xmlfilter">http://javax.xml.transform.sax.SAXTransformerFactory/feature/xmlfilter</a> feature</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><a href="http://javax.xml.XMLConstants/feature/secure-processing">http://javax.xml.XMLConstants/feature/secure-processing</a> feature</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><a href="http://xml.apache.org/xalan/features/incremental">http://xml.apache.org/xalan/features/incremental</a> attribute</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><a href="http://xml.apache.org/xalan/features/optimise">http://xml.apache.org/xalan/features/optimise</a> attribute</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><a href="http://xml.apache.org/xalan/properties/source-location">http://xml.apache.org/xalan/properties/source-location</a> attribute</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>translet-name attribute</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes (with new name)</td>
</tr>
<tr>
<td>destination-directory attribute</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes (with new name)</td>
</tr>
</tbody>
</table>
Table 7. Comparison of the features in the XSLT4J interpreter, the XSLT4J compiler, and the XL TXE-J compiler. (continued)

<table>
<thead>
<tr>
<th>Feature</th>
<th>XSLT4J interpreter (included)</th>
<th>XSLT4J compiler (not included)</th>
<th>XL TXE-J compiler (included)</th>
</tr>
</thead>
<tbody>
<tr>
<td>package-name attribute</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes (with new name)</td>
</tr>
<tr>
<td>jar-name attribute</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes (with new name)</td>
</tr>
<tr>
<td>generate-translet attribute</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes (with new name)</td>
</tr>
<tr>
<td>auto-translet attribute</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes (with new name)</td>
</tr>
<tr>
<td>use-classpath attribute</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes (with new name)</td>
</tr>
<tr>
<td>enable-inlining attribute</td>
<td>No</td>
<td>Yes</td>
<td>No (obsolete in TL TXE-J)</td>
</tr>
<tr>
<td>indent-number attribute</td>
<td>No</td>
<td>Yes</td>
<td>Yes (with new name)</td>
</tr>
<tr>
<td>debug attribute</td>
<td>No</td>
<td>Yes</td>
<td>Yes (with new name)</td>
</tr>
<tr>
<td>Java extensions</td>
<td>Yes</td>
<td>Yes (abbreviated syntax only, xalan:component/xalan:script constructs not supported)</td>
<td>Yes (abbreviated syntax only, xalan:component/xalan:script constructs not supported)</td>
</tr>
<tr>
<td>JavaScript extensions</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Extension elements</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>EXSLT extension functions</td>
<td>Yes</td>
<td>Yes (excluding dynamic)</td>
<td>Yes (excluding dynamic)</td>
</tr>
<tr>
<td>redirect extension</td>
<td>Yes</td>
<td>Yes (excluding redirect:open and redirect:close)</td>
<td>Yes</td>
</tr>
<tr>
<td>output extension</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>nodeset extension</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Nodelist extension functions</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SQL library extension</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>pipeDocument extension</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>evaluate extension</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>tokenize extension</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>XML 1.1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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.xslt.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.xslt.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/java71_[64]/jre/lib.
- Copy the jaxp.properties.sample file to jaxp.properties in /etc/java71_[64]/jre/lib. Uncomment the entries in the jaxp.properties file. Create a symbolic link to the jaxp.properties file from the /usr/java71_[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.
The selective debugging feature, enabled using the command-line option `-XselectiveDebug`, is no longer supported.

More information about problem diagnosis using Java can be found in the Troubleshooting and support.

**Java Debugger (JDB)**

The Java Debugger (JDB) is included in the SDK. The debugger is started with the `jdb` command; it attaches to the JVM using JPDA.

To debug a Java application:

1. Start the JVM with the following options:
   ```java
   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
   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 websites:

- [https://docs.oracle.com/javase/7/docs/technotes/guides/jpda/](https://docs.oracle.com/javase/7/docs/technotes/guides/jpda/)
- [https://docs.oracle.com/javase/7/docs/technotes/guides/jpda/jdb.html](https://docs.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 run time, 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:

```java
System.getProperty("com.ibm.vm.bitmode");
```

Porting Java applications to 64-bit systems

IBM SDK, Java Technology Edition can run in true 64-bit mode on 64-bit systems including IBM Power®, IBM Z and Intel Itanium. By using the IBM SDK, Java applications that run on 32-bit systems can be ported to run on 64-bit systems.

General porting considerations

You will need to consider a number of issues before you port Java applications to 64-bit systems.

- “The IBM implementation of the Java platform on 64-bit processors” on page 168
- “The Java API and porting native code” on page 169
- “JNI and native code” on page 170
- “The advantage of Java programs when moving from 32-bit to 64-bit” on page 170

The IBM implementation of the Java platform on 64-bit processors

IBM provides 64-bit implementations of the Java platform for a number of 64-bit systems, including IBM Power and IBM Z. This is necessary for the following reasons:

- On 32-bit systems, the maximum Java heap space (where Java objects are held at runtime) is 4GB, or 2GB on 31-bit z/OS systems, whereas 64-bit systems can have heap spaces up to (and exceeding) 32GB. This larger heap space is essential for Java applications to be able to use the full capabilities of large, powerful systems.
- Some Java applications use native code through the Java Native Interface (JNI), where the native code is 64-bit. For example, the Java Database Connectivity (JDBC) driver is written in 64-bit native code on a 64-bit platform.
- Some Java applications require other improved capabilities of 64-bit systems.
The Java API and porting native code

Figure 1 is helpful for identifying significant components and for understanding what needs to change when porting applications from 32-bit to 64-bit. All Java applications are written to a defined Java application programming interface (API). This API is the same for the 32-bit and 64-bit versions of the Java platform.

The Java API includes the Java class libraries (for example, awt, swing, net, io, and rmi) that are a fundamental part of the Java platform. The Java API, and all of the Java code, is exactly the same on a 64-bit implementation as it is on a 32-bit implementation. Therefore, if you have Java application code that runs on a 32-bit Java system, it runs unchanged on a 64-bit system.

Many Java applications are not written 100% in the Java language. For these applications, some of their code is written in a non-Java language, such as C or C++, which is generally referred to as “native code”. Porting Java applications that have native code takes more effort.

All Java code (the application and the class libraries) is executed by the Java Virtual Machine (Java VM). The Java VM contains subcomponents that run the Java code and manage the data in the Java Heap. The IBM version of the Java VM includes a just-in-time compiler (JIT) that compiles the Java code into native processor instructions before execution. This compilation process maximises the performance of the code.

In a 64-bit implementation of the Java environment, it is the Java VM that is implemented as a 64-bit program and that is aware of the 64-bit nature of the underlying processor. This includes the JIT compiler, which must compile the Java code to use 64-bit addresses. The JIT compiler is also aware of the full 64-bit instruction set available on the underlying processor.

In IBM’s implementation of the Java platform, the Java VM communicates with the operating system through the Host Porting Interface (HPI). This communication makes the Java VM code independent of the underlying operating system and
allows the IBM implementation of the Java platform to support a range of operating systems on a given processor type.

The SDK is available in 64-bit mode for the following combinations of processor and operating system:

<table>
<thead>
<tr>
<th>Processor type</th>
<th>Operating systems supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Power series</td>
<td>AIX, Linux</td>
</tr>
<tr>
<td>IBM Z</td>
<td>Linux, z/OS</td>
</tr>
<tr>
<td>x86 systems</td>
<td>Windows, Linux</td>
</tr>
</tbody>
</table>

**JNI and native code**

The Java VM, the HPI layer, and the Java code all execute within a single 64-bit operating system process. Normally, there is some native code that executes within that same process. The native code is one or more libraries of code compiled directly for the processor and operating system. The native code is accessed from Java code through the Java Native Interface (JNI). The native code can also access Java objects and methods through the JNI. Some native code is part of the J2SE implementation, such as the code that implements the Java awt classes using the graphics and windowing functions provided by the operating system. Other native code may be provided as part of an application.

There are two other interfaces which are similar to the JNI: the Java Virtual Machine Profiling Interface (JVMPI) and the Java Virtual Machine Debugging Interface (JVMDI). These interfaces provide capabilities for profiling and debugging Java applications. To use these capabilities, you must write some native code.

In the case of the 64-bit implementations of the Java platform, all native code that is associated with the 64-bit Java VM must be 64-bit, because it all runs in the same process space as the Java VM. It is not possible to run 64-bit code and 32-bit code within the same process space because of the incompatibility of the address sizes between these two types of code.

Native code, typically written in the C or C++ languages, is directly aware of the size of addresses. If your application has native code, the native code must be modified to work correctly with the differences in 64-bit environments, such as 64-bit addresses. Finally, native code must also be recompiled and relinked for the 64-bit system.

In summary, the 64-bit Java VM implementation runs on a 64-bit processor and links to code that is also 64-bit “aware”. The Java VM is also able to use the large memory support and 64-bit instructions of the underlying processor. The Java applications it runs are exactly the same as those that can run on 32-bit implementations of Java. Only native code must be ported from 32-bit systems to 64-bit systems.

**The advantage of Java programs when moving from 32-bit to 64-bit**

The Java language and environment make the transition from 32-bit computing to 64-bit computing straightforward. For programs written in other languages that
have explicit address pointers, such as C and C++, porting an application from 32-bit to 64-bit can take considerable effort. This is because every pointer must be redeclared as a 64-bit quantity and any calculation concerning addresses must be checked carefully and adjusted to ensure that it executes correctly in the 64-bit environment. By contrast, 100% Java applications do not even require to be recompiled in order to run correctly in the 64-bit environment. For Java applications, all the hard work is done by the Java VM implementation, while the underlying pointer size is hidden from view.

**Porting native code from 32-bit to 64-bit systems**

If your Java application uses native code, you must first ensure that the native code continues to work correctly when it is ported to the 64-bit environment. You must also make appropriate changes to ensure that the native code works with the 64-bit versions of the JNI, JVMPI, and JVMDI.

Consult the documentation relating to your operating system and to your C/C++ compiler for changes that are needed to port code from 32 bit to 64 bit. In particular, you should note the following differences:

- **Windows**: On 32-bit systems, integers, longs and pointers are all 32 bits. On 64-bit systems, integers and longs remain 32 bits, but pointers become 64 bits and long longs are 64 bits.
- **AIX and Linux**: On 32-bit systems, integers, longs and pointers are all 32 bits. On 64-bit systems, integers remain 32 bits and longs and pointers become 64 bits.

All native code must be adjusted to work with these differences. The best way to do this is to make changes that allow a single copy of the source code to be compiled either for a 32-bit target system or for a 64-bit target system. Although there is no one best way to achieve this goal, a good way to achieve this is to use abstract types for any integers that need to be of the same size as a pointer. For example, when making address calculations, the abstract integer types map either to 32 bits or to 64 bits depending on the size of a pointer.

On AIX and Linux systems, be careful when using long types. If possible, avoid the use of long integers because they change size between 32-bit systems and 64-bit systems.

You must also make sure that data types are correctly aligned. Incorrect alignment can cause serious loss of code performance. The correct alignment for various data types is shown in the following table:

<table>
<thead>
<tr>
<th>C/C++ data types</th>
<th>AIX, Linux, z/OS</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32 bit (31-bit z/OS)</td>
<td>64 bit</td>
</tr>
<tr>
<td>char</td>
<td>size/alignment</td>
<td>size/alignment</td>
</tr>
<tr>
<td>short</td>
<td>1/1</td>
<td>1/1</td>
</tr>
<tr>
<td>int</td>
<td>2/2</td>
<td>2/2</td>
</tr>
<tr>
<td>long</td>
<td>4/4</td>
<td>4/4</td>
</tr>
<tr>
<td>long long</td>
<td>8/8</td>
<td>8/8</td>
</tr>
<tr>
<td>pointer</td>
<td>4/4</td>
<td>8/8</td>
</tr>
</tbody>
</table>
When the native code has been adjusted to take account of the differences between 32-bit systems and 64-bit systems, tune the code to accommodate differences in the interfaces to the Java VM: the JNI, the JVMDI, and the JVMPI.

**JNI considerations**

If you use JNI from native code on a 64-bit system, you must ensure that length-related quantities in JNI are defined as 64 bit.

You must ensure that length-related quantities in JNI are defined as 64 bit even though the underlying objects and quantities within the Java Developer Kit (JDK) are restricted to values no greater than 32 bit. An error occurs if native code uses values greater than 32 bit for these quantities.

The definition of JNI is held in two header files (jni.h and jni_md.h), which are included by any native code that uses JNI.

In a 64-bit version of the JDK, the JNI interface itself changes as follows:

**Pointers (foo *)**

All pointers become 64 bit.

<jni_md.h> jint

This is defined as a 32-bit value (int), since on Unix systems a `long` is a 64-bit integer.

<jni.h> jsize

This type is used extensively for quantities that imply lengths. This is changed on 64-bit systems to a `jlong` so that it is 64 bits, in contrast to the definition on 32-bit systems where it is a `jint`, which is 32 bits.

jfloat, jdouble

Values remain the same as in 32-bit systems.

jobject

Defined as a pointer and so is 64 bit.

jclass, jthrowable, jstring, jarray

All declared as `jobject` and are all 64 bit.

jvalue A union of all basic 'j' types and is already 64 bit.

jmethodID, jfieldID

Declared as pointer types and are 64-bit quantities.

JNINativeMethod

struct is 3 pointers and becomes 24 bytes.

JNIEnv Declared as a pointer type and is 64 bit.
JavaVM  Declared as a pointer type and is 64 bit.

JNI Native Interface
Declared as an array of pointers and these are all 64 bit.

DefineClass
Has a buffer parameter with an associated size parameter. The size parameter is declared as a jsize and therefore is a 64-bit integer length. However, class objects cannot be this large, so there is an upper bound to the value of the size parameter and is less than 32 bits.

PushLocalFrame
Has a capacity defined by a jint and is limited to 32 bits.

EnsureLocalCapacity
Has a capacity defined by a jint and is limited to 32 bits.

NewString
Has a jsize length parameter and becomes 64 bit. However, the size of the supplied unicode string has a maximum length no greater than a 32-bit integer.

GetStringLength
Returns a jsize, which is 64 bit. The length of a Java string is limited to 32-bit integer values.

GetStringUTFLength
Returns a jsize, which is 64 bit. The length of a Java string is limited to 32-bit integer values.

GetArrayLength
Returns a jsize, which is 64 bit. However, Java arrays cannot have a length greater than a 32-bit integer value.

NewXxxArray
(where Xxx is the type of a primitive: for example, NewBooleanArray.) All have jsize length parameters, which are 64 bit. Java arrays cannot have a length greater than a 32-bit integer value.

GetObjectArrayElement, SetObjectArrayElement
Both have jsize index values which are 64 bit. Java arrays cannot have a length greater than a 32-bit integer.

GetXxxArrayRegion, SetXxxArrayRegion
(where Xxx is the type of a primitive: for example, GetIntArrayRegion, SetFloatArrayRegion.) All have jsize start and length fields, which are 64 bit. Java arrays cannot have a length greater than 32-bit integer value.

RegisterNatives
Has a jint parameter - number of methods.

GetStringRegion, GetStringUTFRegion
Both have start and length parameters as jsize, which are 64 bit. Strings are limited to 32-bit lengths.

JavaVMOption
struct has two pointers: both become 64 bit.

JavaVMIInitArgs
The alignment of this struct is affected by change to JavaVMOption struct.

JavaVMAAttachArgs
Alignment is affected by two pointers.
**Usage of the JNI interface**

Beyond the form of the JNI interface, you must consider how the JNI is used by native libraries on a 64-bit system.

There are a number of issues to consider:

- On some 64-bit systems, certain native objects that are passed across the JNI are 64 bit, whereas they are 32 bit on equivalent 32-bit systems. Examples of this type of object are Socket handles and File handles.
- Passing 64-bit objects across the JNI is not a problem. However, in many cases, these objects are stored in Java object fields. The implication is that the type of these object fields changes on a 64-bit system compared with existing 32-bit systems. You must change your Java code to achieve this.
- If the size of a Java object field is different on the 64-bit platform from the 32-bit platform, any native code that is working with that object field must be changed to reflect the difference in size when the code is ported from 32 bit to 64 bit. You can best deal with this by having a full description of each of the Java object types that is affected in this way. If your application code stores "native data" within Java objects, then you must modify your Java code to ensure that there is sufficient space within your Java objects to store all of the native data.

**JVMPi and JVMDI interfaces**

If your application has native code that uses the Java VM Profiling Interface (JVMPi) or the Java VM Debug Interface (JVMDI), these interfaces are affected by the change from 32 bit to 64 bit.

**JVMPi interface**

The JVMPi interface is defined in the header file jvmpi.h.

The following changes have been made to the interface definition:

- struct {
  unsigned char *class_data; /* content of class file */
  jint class_data_len; /* class file length */
  unsigned char *new_class_data; /* instrumented class file */
  jint new_class_data_len; /* new class file length */
  void *(*malloc_f)(size_t); /* memory allocation function */
  } class_load_hook;

This definition is the same as the definition on IBM 32-bit implementations. However, it differs from the Sun specification in relation to the definition of the memory allocation function. Defining the memory allocation function with a size_t type parameter actually enables this structure to work unchanged on 64-bit systems, because size_t becomes 64 bits.

- struct {
  jint arena_id;
  jobjectID class_id; /* id of object class */
  jint is_array; /* JVMPi_NORMAL_OBJECT, ... */
  jsize size; /* size in number of bytes */
  jobjectID obj_id; /* id assigned to this object */
  } obj_alloc;

- struct {
  jsize data_len;
  char *data;
  } object_dump;

Here, the data_len field becomes jsize in type, which is 64 bits on 64-bit systems, but 32 bits on 32-bit systems.
Remember that the JVMPi interface is typically extended beyond its basic set of functions, in both IBM 64-bit implementations and IBM 32-bit implementations.

**JVMDI interface**

The JVMDI interface is defined in the header file `jvmdi.h`.

The JVMDI interface has no changes for 64-bit, other than changing Pointer parameters from 32 bit to 64 bit. All JVMDI structures and function call definitions remain the same in 64 bit as they are in 32 bit.

---

**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 9 on page 176](#) 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 9. Signals used by the JVM

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Signal type</th>
<th>Description</th>
<th>Disabled by -Xrs</th>
<th>Disabled by -Xrs:sync</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGBUS (7)</td>
<td>Exception</td>
<td>Incorrect access to memory (data misalignment)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SIGSEGV (11)</td>
<td>Exception</td>
<td>Incorrect access to memory (write to inaccessible memory)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SIGILL (4)</td>
<td>Exception</td>
<td>Illegal instruction (attempt to call an unknown machine instruction)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SIGFPE (8)</td>
<td>Exception</td>
<td>Floating point exception (divide by zero)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SIGABRT (6)</td>
<td>Error</td>
<td>Abnormal termination. The JVM raises this signal whenever it detects a JVM fault.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SIGINT (2)</td>
<td>Interrupt</td>
<td>Interactive attention (CTRL-C). JVM exits normally.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SIGTERM (15)</td>
<td>Interrupt</td>
<td>Termination request. JVM will exit normally.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SIGHUP (1)</td>
<td>Interrupt</td>
<td>Hang up. JVM exits normally.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SIGQUIT (3)</td>
<td>Control</td>
<td>By default, this triggers a Javadump.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>No Name (40)</td>
<td>Control</td>
<td>An AIX reserved signal. Used by the AIX JVM for internal control purposes.</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 9. Signals used by the JVM (continued)

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Signal type</th>
<th>Description</th>
<th>Disabled by -Xrs</th>
<th>Disabled by -Xrs:sync</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGRECONFIG (58)</td>
<td>Control</td>
<td>Reserved to detect any change in the number of CPUs, processing capacity, or physical memory.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SIGTRAP (5)</td>
<td>Control</td>
<td>Used by the JIT.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SIGRTMIN (50)</td>
<td>Control</td>
<td>Used by the JVM for internal control purposes.</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SIGRTMAX (57)</td>
<td>Control</td>
<td>Used by the SDK.</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SIGCHLD (20)</td>
<td>Control</td>
<td>Used by the SDK for internal control.</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

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 -qf1trap 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 install_dir

To use libjsig.a:
• Link it with the application that creates or embeds a JVM:
  cc_r -q64 -Linstall_dir -ljsig -Linstall_dir/jre/bin/j9vm -ljvm java_application.c

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

Writing JNI applications

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

Restriction: Version 1.1 of the JNI is not supported.

This version number determines only the level of the JNI to use. The actual level
of the JVM that is created is specified by the JSE libraries (use the java -version
command to show the JVM level). The JNI level does not affect the language
specification that is implemented by the JVM, the class library APIs, or any other
area of JVM behavior. For more information, see
https://docs.oracle.com/javase/7/docs/technotes/guides/jni/.

If your application needs two JNI libraries, one built for 32-bit and the other for
64-bit, use the com.ibm.vm.bitmode system property to determine if you are
running with a 32-bit 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 V12.1 is supported for AIX 64-bit on IBM POWER.

JNI compatibility

If you are working with the Java Native Interface (JNI), you must ensure that your
system is set up appropriately.

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:

```bash
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** variable must include the directories that contain the JVM shared libraries, `lib_dir` and `lib_dir/j9vm`, as well as the directories that contain the application’s shared libraries.

You must build:

- 32-bit and 64-bit executables and shared objects. To do this, use the `-qarch=ppc` option.
- JNI executables and shared objects. For the 32-bit SDK, these should be built as 32-bit executables or shared objects. For the 64-bit SDK, they should be built as 64-bit programs or shared objects, using the `-q64` option.

For information on JNI runtime linking, see [JNI runtime linking](#).

### JNI runtime linking

The Java Native Interface (JNI) enables runtime linking to dynamic native libraries.

For runtime linking, applications can be linked by using the `-brtl` loader option. If runtime linking causes a symbol conflict, the application must resolve the conflict by renaming the symbol on the application side, or by turning off runtime linking.

### Dynamic linking

To dynamically link a native library, you should 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 -qnkshrobj [-qarch=ppc | -q64] [-I install_dir] include
-o libnm.a nm.c
```

The `-qnkshrobj` option disables runtime linking. For more information 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](http://www.redbooks.ibm.com/abstracts/sg245674.html)
- The AIX online documentation at [https://www.ibm.com/support/knowledgecenter/ssw_aix/welcome](https://www.ibm.com/support/knowledgecenter/ssw_aix/welcome)

Before you run a Java program that uses native methods, ensure that **LIBPATH** contains the list of directories that hold the shared objects for the native methods. For more information about building AIX shared objects, see [C and C++ Application Development on AIX](https://www.ibm.com/redbooks). Go to [https://www.ibm.com/redbooks](https://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 information 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 you build 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 do the following tasks:

- Add /usr/lib and /lib to the list of directories that are searched for shared objects. All programs need shared objects that are stored in these directories.
- Add lib_dir and lib_dir/j9vm to the list of directories that are searched for shared objects. These directories contain the Java SDK shared libraries. You also want to link with libjvm.so (by using the -ljvm option).

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

```bash
cc_r [-qarch=ppc | -q64] -Iinstall_dir/include -o invAPITest -L/usr/lib -L/lib -Llib_dir/j9vm -Llib_dir -ljvm invAPITest.c
```

When you run 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 JVM to find your class files. If you modify the Java 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 following command:

```
dump -ov <program name>
```

The following output is generated:

```
***Object Module Header***
# Sections Symbol Ptr # Symbols OptHdr 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>
```

The following output is generated:
>dump -X64 -ov <program name>

***Object Module Header***

<table>
<thead>
<tr>
<th># Sections</th>
<th>Symbol Ptr</th>
<th># Symbols</th>
<th>Opt Hdr</th>
<th>Len</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0x0001a728</td>
<td>1305</td>
<td>120</td>
<td>0x1002</td>
<td></td>
</tr>
</tbody>
</table>

Flags=( EXEC DYNLOAD DEP_SYSTEM )
Timestamp = "14 Oct 03:26:43 2005"
Magic = 0xf7 (64-bit XCOFF)

***Optional Header***

<table>
<thead>
<tr>
<th>Tsize</th>
<th>Dsize</th>
<th>Bsize</th>
<th>Tstart</th>
<th>Dstart</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00127f8</td>
<td>0x00001b80</td>
<td>0x00000470</td>
<td>0x1000001f8</td>
<td>0x1100009f0</td>
</tr>
</tbody>
</table>

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 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 approach is on a 1-to-1 mapping with the kernel threads. When you develop a JNI program, you must run with a 1-to-1 thread model and system contention scope if you create pthreads in your own program. You can control this behavior by using the following environment setting:

export AIXTHREAD_SCOPE=S

Another option is to preset the thread’s scope attribute to PTHREAD_SCOPE_SYSTEM by using the AIX pthread_attr_setscope function when the thread is created.

You can store native methods as follows:

**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 -l<name> 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 -l<name> option.

Programs can also link dynamically to shared libraries and shared objects, for example by 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:

```java
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:

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

**Note:** To ensure that dynamic linking of native libraries works successfully you can, optionally, implement the lifecycle functions JNI_Onload() and JNI_OnUnload() in the library. If you have implemented JNI_Onload(), the native library must export it otherwise it is not visible to the runtime, and the JVM assumes that the library requires only the JNI version JNI_VERSION_1.1. If JNI_OnUnload() has been implemented, it must also be exported. If JNI_Onload() is implemented and exported, then the latest JNI version is returned; for example, JNI_VERSION_1.8.

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

   ```java
   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`.
   ```bash
   javac Sample.java
   ```

3. Use `javah` to create a header file for the native code.
   ```bash
   javah Sample
   ```

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

---

```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 -includeinstall_dir Sample.c
   -o libSample.o -q64
   ```

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

7. Run the Sample class.
   ```
   LIBPATH=. java Sample
   ```
   or
   ```
   java -Djava.library.path=. 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.

API documentation to support the package containing these classes is available here: [API documentation]

**CORBA support**

The Java Platform, Standard Edition (JSE) supports, at a minimum, the specifications that are defined in the compliance document from Oracle. In some cases, the IBM JSE ORB supports more recent versions of the specifications.

The minimum specifications supported are defined in the Official Specifications for CORBA support in Java SE 6: [https://docs.oracle.com/javase/7/docs/api/org.omg/CORBA/doc-files/compliance.html](https://docs.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.


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:


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:

[https://www.omg.org/cgi-bin/doc?ptc/00-08-07](https://www.omg.org/cgi-bin/doc?ptc/00-08-07)

The default port that is used by the Transient Name Server (the tnameserv command), when no ORBInitialPort parameter is given, has changed from 900 to 2809, which is the port number that is registered with the IANA (Internet Assigned Number Authority) for a CORBA Naming Service. Programs that depend on this default might have to be updated to work with this version.

The initial context that is returned from the Transient Name Server is now an org.omg.CosNaming.NamingContextExt. Existing programs that narrow the reference to a context org.omg.CosNaming.NamingContext still work, and do not need to be recompiled.

The ORB supports the -ORBInitRef and -ORBDefaultInitRef parameters that are defined by the Interoperable Naming Service specification, and the
ORB::string_to_object operation now supports the ObjectURL string formats (corbaloc: and corbaname:) that are defined by the Interoperable Naming Service specification.

The OMG specifies a method ORB::register_initial_reference to register a service with the Interoperable Naming Service. However, this method is not available in the Oracle Java Core API at this release. Programs that have to register a service in the current version must invoke this method on the IBM internal ORB implementation class. For example, to register a service “MyService”:

```
((com.ibm.CORBA.iiop.ORB)orb).register_initial_reference("MyService", serviceRef);
```

Where orb is an instance of org.omg.CORBA.ORB, which is returned from ORB.init(), and serviceRef is a CORBA Object, which is connected to the ORB. This mechanism is an interim one, and is not compatible with future versions or portable to non-IBM ORBs.

**System properties for tracing the ORB**

A runtime debug feature provides improved serviceability. You might find it useful for problem diagnosis or it might be requested by IBM service personnel.

**Tracing Properties**

- **com.ibm.CORBA.Debug=true**
  
  Turns on ORB tracing.

- **com.ibm.CORBA.CommTrace=true**
  
  Adds GIOP messages (sent and received) to the trace.

- **com.ibm.CORBA.Debug.Output=<file>**
  
  Specify the trace output file. By default, this is of the form orbtrc.DDMMYYYY.HHmm.SS.txt.

**Example of ORB tracing**

For example, to trace events and formatted GIOP messages from the command line, type:

```
java -Dcom.ibm.CORBA.Debug=true
     -Dcom.ibm.CORBA.CommTrace=true <myapp>
```

**Limitations**

Do not enable tracing for normal operation, because it might cause performance degradation. Even if you have switched off tracing, FFDC (First Failure Data Capture) is still working, so serious errors are reported. If a debug output file is generated, examine it to check on the problem. For example, the server might have stopped without performing an ORB.shutdown().

The content and format of the trace output might vary from version to version.

**System properties for tuning the ORB**

The ORB can be tuned to work well with your specific network. The properties required to tune the ORB are described here.

- **com.ibm.CORBA.FragmentSize=<size in bytes>**
  
  Used to control GIOP 1.2 fragmentation. The default size is 1024 bytes in the initial release of this version, and 4096 bytes in service refresh 1 and later.
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 10. Methods affected when running with Java SecurityManager**

<table>
<thead>
<tr>
<th>Class/Interface</th>
<th>Method</th>
<th>Required permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.omg.CORBA.ORB</td>
<td>init</td>
<td>java.net.SocketPermission resolve</td>
</tr>
<tr>
<td>org.omg.CORBA.ORB</td>
<td>connect</td>
<td>java.net.SocketPermission listen</td>
</tr>
<tr>
<td>org.omg.CORBA.ORB</td>
<td>resolve_initial_references</td>
<td>java.net.SocketPermission connect</td>
</tr>
<tr>
<td>org.omg.CORBA.portable.ObjectImpl</td>
<td>_is_a</td>
<td>java.net.SocketPermission connect</td>
</tr>
<tr>
<td>org.omg.CORBA.portable.ObjectImpl</td>
<td>_non_existent</td>
<td>java.net.SocketPermission connect</td>
</tr>
<tr>
<td>org.omg.CORBA.portable.ObjectImpl</td>
<td>OutputStream _request (String, boolean)</td>
<td>java.net.SocketPermission connect</td>
</tr>
<tr>
<td>org.omg.CORBA.portable.ObjectImpl</td>
<td>_get_interface_def</td>
<td>java.net.SocketPermission connect</td>
</tr>
<tr>
<td>org.omg.CORBA.Request</td>
<td>invoke</td>
<td>java.net.SocketPermission connect</td>
</tr>
<tr>
<td>org.omg.CORBA.Request</td>
<td>send_deferred</td>
<td>java.net.SocketPermission connect</td>
</tr>
<tr>
<td>org.omg.CORBA.Request</td>
<td>send_oneway</td>
<td>java.net.SocketPermission connect</td>
</tr>
<tr>
<td>javax.rmi.PortableRemoteObject</td>
<td>narrow</td>
<td>java.net.SocketPermission connect</td>
</tr>
</tbody>
</table>

### 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
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. [https://www.omg.org/cgi-bin/doc?ptc/00-01-06.pdf](https://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 websites related to RMI and related technologies.

Here are some sites to help you with this technology:

- The RMI trail in the Java Tutorial: [https://docs.oracle.com/javase/tutorial/rmi/](https://docs.oracle.com/javase/tutorial/rmi/)
- The RMI API Javadoc HTML contains the most up-to-date RMI API documentation: [https://docs.oracle.com/javase/7/docs/api/java/rmi/package-summary.html](https://docs.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: [https://docs.oracle.com/javase/7/docs/technotes/guides/idl/index.html](https://docs.oracle.com/javase/7/docs/technotes/guides/idl/index.html)
- The Java IDL Trail in the Java Tutorial: [https://docs.oracle.com/javase/7/docs/technotes/guides/idl/GShome.html](https://docs.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: https://docs.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 IDL. It defines the way bits are sent over a wire between CORBA clients and servers. CORBA is a standard distributed object architecture developed by the Object Management Group (OMG). Interfaces to remote objects are described in a platform-neutral interface definition language (IDL). Mappings from IDL to specific programming languages are implemented, binding the language to CORBA/IIOP.

The Java Standard Edition CORBA/IIOP implementation is known as Java IDL. Along with the IDL to Java (idlj) compiler, Java IDL can be used to define, implement, and access CORBA objects from the Java programming language.

The Java IDL Web page: [Java IDL (CORBA)] 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: Getting Started with Java IDL.

**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 193), 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: https://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: [https://www.omg.org/cgi-bin/doc?formal/01-06-07](https://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: [https://www.omg.org/cgi-bin/doc?formal/01-06-07](https://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: rmic - The Java RMI Compiler
• Windows version: rmic - The Java RMI Compiler

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 idlj - The IDL-to-Java Compiler

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:

```java
import java.rmi.*;
...
Naming.rebind("MyObject", myObj);
```

use:

```java
import javax.naming.*;
...
names.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:

   ```java
   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:

   ```java
   FileInputStream myFile = new FileInputStream("t.tmp");
   ObjectInputStream myStream = new ObjectInputStream(myFile);
   Wombat myWombat = (Wombat)myStream.readObject();
   org.omg.CORBA.ORB myORB = org.omg.CORBA.ORB.init(new String[0], null);
   ((javax.rmi.CORBA.Stub)myWombat).connect(myORB);
   // myWombat is now connected to myORB. To connect other objects to the
   // same ORB, use PortableRemoteObject.connect(nextWombat, myWombat);
   ```

The JNDI approach is much simpler, so it is preferable to use it whenever possible.

3. Either change your remote implementation classes to inherit from javax.rmi.PortableRemoteObject, or explicitly to export implementation objects after creation by calling PortableRemoteObject.exportObject(). For more discussion on this topic, read "Connecting IIOP stubs to the ORB" on page 192.

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 ([https://www.omg.org/cgi-bin/doc?formal/01-06-07](https://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
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: https://www.omg.org/cgi-bin/doc?formal/01-06-07
- Do not inherit the same method name into a remote interface more than once from different base remote interfaces.
- Be careful when using names that are identical other than their case. The use of a type name and a variable of that type with a name that differs from the type name in case only is supported. Most other combinations of names that are identical other than their case are not supported.
- Do not depend on run time sharing of object references to be preserved exactly when transmitting object references to IIOP. Runtime sharing of other objects is preserved correctly.
- Do not use the following features of RMI, which do not work in RMI-IIOP:
  - RMISocketFactory
  - UnicastRemoteObject
  - Unreferenced
  - The Distributed Garbage Collector (DGC) interfaces

Additional information

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

Servers must be thread safe

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

Interoperating with other ORBs

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

Note: Although ORBs written in different languages should be able to interoperate, the Java ORB has not been fully tested with other vendors' ORBs.
When do I use UnicastRemoteObject vs PortableRemoteObject?

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

**Known limitations**

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

Implementing the Connection Handler Pool for RMI

Thread pooling for RMI Connection Handlers is not enabled by default.

**About this task**

To enable the connection pooling implemented at the RMI TCPTransport level, set the option

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

This version of the Runtime Environment does not have a setting that you can use to limit the number of threads in the connection pool.

**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 6.1 thread scheduling, see:

- `pthread_setschedparam Subroutine` and [Scheduling Threads](#)

**Developing Java applications**

The SDK contains many tools and libraries required for Java software development.

See [IBM Software Developers Kit (SDK)](#) on page 2 for details of the tools available.

**Support for XToolkit**

XToolkit is included by default. You need XToolkit when using the SWT_AWT bridge in Eclipse to build an application that uses both SWT and Swing.

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

Related links:
• An example of integrating Swing into Eclipse RCPs: [https://eclipsezone.com/eclipse/forums/t45697.html](https://eclipsezone.com/eclipse/forums/t45697.html)


• Set up information is available on the Oracle Corporation Web site: [https://docs.oracle.com/javase/7/docs/technotes/guides/awt/1.5/xawt.html](https://docs.oracle.com/javase/7/docs/technotes/guides/awt/1.5/xawt.html)

---

**Support for the Java Attach API**

Your application can connect to another “target” virtual machine using the Java Attach API. Your application can then load an agent application into the target virtual machine, for example to perform tasks such as monitoring status.

Code for agent applications, such as JMX agents or JVMTI agents, is normally loaded during virtual machine startup by specifying special startup parameters. Requiring startup parameters might not be convenient for using agents on applications that are already running, such as WebSphere Application Servers. You can use the Java Attach API to load an agent at any time, by specifying the process ID of the target virtual machine. The Attach API capability is sometimes called the “late attach” capability.

Support for the Attach API is enabled by default.

**Security considerations**

Security for the Java Attach API is handled by POSIX file permissions.

The Java Attach API creates files and directories in a common directory.

The key security features of the Java Attach API are:

• A process using the Java Attach API must be owned by the same UNIX user ID as the target process. This constraint ensures that only the target process owner or root can attach other applications to the target process.

• The common directory uses the sticky bit to prevent a user from deleting or replacing a subdirectory belonging to another user. To preserve the security of this mechanism, set the ownership of the common directory to ROOT. This directory will contain files such as _attachlock, _master, and _notifier, which are used only for synchronization. These files can be owned by any user, and must have read and write permission. However, you can remove execute permission on these files, if present. The files are empty and will be re-created automatically if deleted.

• The files in the subdirectory for a process, with the exception of a lock file, are accessible only by the owner of a process. The subdirectory has owner read, write, and execute permissions plus group and world execute permissions. In this directory, read and write access are restricted to the owner only, except for the attachNotificationSync file, which must have world and group write permissions. This exception does not affect security because the file is used exclusively for synchronization and is never written to or read.

• Information about the target process can be written and read only by the owner.

• Java 5 SR10 allowed users in the same group to access to each others’ processes. This capability was removed in later versions.
You must secure access to the Java Attach API capability to ensure that only authorized users or processes can connect to another virtual machine. If you do not intend to use the Java Attach API capability, disable this feature using a Java system property. Set the `com.ibm.tools.attach.enable` system property to the value `no`; for example:

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

The Attach API can be enabled by setting the `com.ibm.tools.attach.enable` system property to the value `yes`; for example:

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

### Using the Java Attach API

By default, the target virtual machine is identified by its process ID. To use a different target, change the system property `com.ibm.tools.attach.id`; for example:

```
-Dcom.ibm.tools.attach.id=<process_ID>
```

The target process also has a human-readable “display name”. By default, the display name is the command line used to start Java. To change the default display name, use the `com.ibm.tools.attach.displayName` system property. The ID and display name cannot be changed after the application has started.

The Attach API creates working files in a common directory, which by default is called `.com_ibm_tools_attach` and is created in the system temporary directory. The system property `java.io.tmpdir` holds the value of the system temporary directory. On non-Windows systems, the system temporary directory is typically `/tmp`.

You can specify a different common directory from the default, by using the following Java system property:

```
-Dcom.ibm.tools.attach.directory=directory_name
```

This system property causes the specified directory, `directory_name`, to be used as the common directory. If the directory does not already exist, it is created, however the parent directory must already exist, and be on a locally-mounted file system. Using a common directory on a network file system causes errors. For example, the following system property creates a common directory called `myattachapidir` in the `usr` directory. The `usr` directory must already exist on a locally-mounted file system.

```
-Dcom.ibm.tools.attach.directory=/usr/myattachapidir
```

The common directory must be located on a local drive; specifying a network mounted file system causes errors.

If your Java application ends abnormally, for example, following a crash or a SIGKILL signal, the process subdir directory is not deleted. The Java VM detects and removes obsolete subdirectories where possible. The subdir directory can also be deleted by the owning user ID.

On heavily loaded system, applications might experience timeouts when attempting to connect to target applications. The default timeout is 120 seconds. Use the `com.ibm.tools.attach.timeout` system property to specify a different timeout value in milliseconds. For example, to timeout after 60 seconds:

```
-Dcom.ibm.tools.attach.timeout=60000
```
A timeout value of zero indicates an indefinite wait.

For JMX applications, you can disable authentication by editing the `<JAVA_HOME>/jre/lib/management/management.properties` file. Set the following properties to disable authentication in JMX:

- `com.sun.management.jmxremote.authenticate=false`
- `com.sun.management.jmxremote.ssl=false`

Problems with the Attach API result in one of the following exceptions:

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

A useful reference for information about the Attach API can be found at [https://docs.oracle.com/javase/7/docs/technotes/guides/attach/index.html](https://docs.oracle.com/javase/7/docs/technotes/guides/attach/index.html). The IBM implementation of the Attach API is equivalent to the Oracle Corporation implementation. However, the IBM implementation cannot be used to attach to, or accept attach requests from, non-IBM virtual machines. To use the attach API to attach to target processes from your application, you must add the tools.jar library to the application classpath. This library is not required for the target processes to accept attach requests.

Related reference:

```
-Dcom.ibm.tools.attach.enable` on page 536
```

Enable the Attach API for this application.

Related information:

```
“Attach API problem determination” on page 308
```

Common problems and explanations that can help you solve issues that involve the Attach API.

---

## The Data Access Accelerator Library

You can use the methods in the Data Access Accelerator (DAA) library to improve performance when your Java code manipulates native data.

The Java language does not normally support operations on native data structures (primitive types and binary coded decimals within Java byte arrays and streams). Instead, these structures are converted into Java objects before the operations are carried out. For example, binary-coded decimals are not native types in Java and they are often converted into `BigDecimal` or `BigInteger` Java objects. This processing takes time and resources, and puts pressure on the Java heap. These levels of indirection also make it difficult for Java applications to use any hardware acceleration that is available on the underlying platform.

If your application manipulates native data, you can use the Data Access Accelerator (DAA) library in the `com.ibm.dataaccess` package to improve performance. When you use DAA methods in your application code, the IBM J9 VM automatically exploits any available hardware features that accelerate the operation, depending on your operating system, to optimize the manipulation of primitive data types and binary coded decimals within Java byte arrays and streams. For example, you can use DAA APIs to efficiently marshal multi-byte datatypes such as `int` and `double` from byte arrays, or to manipulate packed-decimal data that is stored within COBOL copybooks.
By using the APIs provided, your application can therefore gain a number of advantages:

- avoid the need for unnecessary object creation and intermediate processing
- use available hardware acceleration
- remain platform-independent

**DAA functions**

The com.ibm.dataaccess package provides the following functions:

- Packed decimal operations:
  - Arithmetic operations
  - Relation operations
  - Conversions to and from other decimal types, such as external decimal and Unicode decimal, and primitive types
  - Validity checking
  - Shifting and moving operations
- BigDecimal and BigInteger conversions from and to external decimal and Unicode decimal types
- Marshalling operations: marshalling and unmarshalling primitive types (short, int, long, float, and double) to and from byte arrays

For more information, see the API documentation for the com.ibm.dataaccess package.

**Tip:** Data access acceleration is available from Version 7 Release 1. If you are creating an application that might run on multiple versions of the product, write code that can detect whether data access acceleration is supported, then use that support if it is available. For example, create an inner class called IBMDataAccessUtility, within the class that uses the Data Access Accelerator API. Create a static final boolean variable within the inner class, and initialize the variable by querying the availability of the Data Access Accelerator APIs. You can then use the variable to decide at run time whether to call a Data Access Accelerator API or use an alternative approach to deal with native data types.

**Example: Converting a packed decimal to a BigDecimal object**

In this example, a signed packed decimal of 100 digits (not including the sign), is stored in a byte array called pdarray.

Normal code:

```java
... String pdsb = new StringBuffer(); int precision = 100; if (isNegative(pdarray, precision)) pdsb.append('-'); for (int i = 0; i < precision; ++i) pdsb.append(getDigit(pdarray, i) + '0'); return new BigDecimal(pdsb.toString()); ...
```

Equivalent code, which uses the DAA API:

```java
... return com.ibm.dataaccess.DecimalData.convertPackedDecimalToBigDecimal(pdarray, 0, precision, 0, false); ...
```
Example: Adding two packed decimal types

In this example, two packed decimals, pdarray1 and pdarray2, are added, and the result, pdarrayResult, is returned. The packed decimals, and the result, are stored as byte arrays. The precision (the number of decimal digits in the decimal value, not including the sign) of the two initial arrays is 100.

Without using the Data Access Accelerator API, you have to write your own methods to convert between packed decimal and BigDecimal formats (for example, by using an intermediate String object):

```java
BigDecimal operand1 = myPackedDecimalToBigDecimalMethod(pdarray1, 100);
BigDecimal operand2 = myPackedDecimalToBigDecimalMethod(pdarray2, 100);

BigDecimal result = operand1.add(operand2);
myBigDecimalToPackedDecimalMethod(result, pdarrayResult);
return pdarrayResult;
```

Equivalent code, which uses the DAA API. The precision of the result is 1 greater than the precision of the two initial decimals. All the packed decimals have an offset (where the packed decimal begins) of 0:

```java
com.ibm.dataaccess.DecimalData.addPackedDecimal(pdarrayResult, 0, 101, pdarray1, 0, 100, pdarray2, 0, 100, false);
return pdarrayResult;
```
Chapter 6. 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.

Important: As announced in the Oracle Java SE Support roadmap support for Java plug-in and Web Start technology in Java SE 7 ends July 2016. Accordingly, support is also removed from IBM SDK, Java Technology Edition, Version 7 Release 1. For continued support, move to the latest release of the IBM SDK for all applications launched or deployed through web browsers.

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. Support is no longer available for the plug-in from IBM SDK, Java Technology Edition, Version 7 Release 1 Service Refresh 3 Fix Pack 50 and later updates.

Important: As announced in the Oracle Java SE Support roadmap support for Java plug-in technology in Java SE 7 ends in July 2016. Accordingly, support is also removed from IBM SDK, Java Technology Edition, Version 7 Release 1. For continued support, move to the latest release of the IBM SDK for all applications launched or deployed through web browsers.

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: https://docs.oracle.com/javase/7/docs/technotes/guides/jweb/applet/applet_dev_guide.html.

Note: The Classic plug-in, sometimes referred to as the First-generation plug-in, was deprecated in IBM SDK, Java Technology Edition, Version 7. This plug-in will not be supported in future releases. Instead, use the Next-generation plug-in to run applets in the browser.

Supported browsers

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

Important: In line with Oracle changes, support is no longer available for the Java plug-in from IBM SDK, Java Technology Edition, Version 7 Release 1. For continued support, move to the latest release of the IBM SDK for all applications launched or deployed through web browsers.

Table 11. Browsers supported by the Java plug-in on 64-bit AIX

<table>
<thead>
<tr>
<th>Browser</th>
<th>Supported Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox</td>
<td>3.6.25.3. Available at <a href="https://www.ibm.com/servers/aix/browsers/">https://www.ibm.com/servers/aix/browsers/</a></td>
</tr>
</tbody>
</table>

Related information:
Installing the Java plug-in

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

**Important:** In line with Oracle changes, support is no longer available for the Java plug-in from IBM SDK, Java Technology Edition, Version 7 Release 1. For continued support, move to the latest release of the IBM SDK for all applications launched or deployed through web browsers.

The Java plug-in is based on the Mozilla [Open JVM Integration](https://openjdk.java.net/projects/jigsaw/document.html) 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>-71/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>-71/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.

**Before you begin**

**Important:** In line with Oracle changes, support is no longer available for the Java plug-in from IBM SDK, Java Technology Edition, Version 7 Release 1. For continued support, move to the latest release of the IBM SDK for all applications launched or deployed through web browsers.

**About this task**

To start this Java application, run the script:

```
/usr/java71[_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

**Important:** In line with Oracle changes, support is no longer available for the Java plug-in from IBM SDK, Java Technology Edition, Version 7 Release 1. For continued support, move to the latest release of the IBM SDK for all applications launched or deployed through web browsers.

### 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.

### Before you begin

**Important:** In line with Oracle changes, support is no longer available for the Java plug-in from IBM SDK, Java Technology Edition, Version 7 Release 1. For continued support, move to the latest release of the IBM SDK for all applications launched or deployed through web browsers.

### About this task

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

```html
<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:

  ```bash
  appletviewer <name>.
  ```

  `<name>` is one of the following options:

  - 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:

  ```bash
  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://mywebpage.com/demo/applets/MyApplet/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
  ```

## 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. Support is no longer available for Web Start from IBM SDK, Java Technology Edition, Version 7 Release 1 Service Refresh 3 Fix Pack 50 and later updates.

**Important:** As announced in the [Oracle Java SE Support roadmap](https://docs.oracle.com/javase/7/docs/technotes/guides/javaws/developersguide/syntax.html#resources) support for Java Web Start technology in Java SE 7 ends in July 2016. Accordingly, support is also removed from IBM SDK, Java Technology Edition, Version 7 Release 1. For continued support, move to the latest release of the IBM SDK for all applications launched or deployed through web browsers.

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 [https://docs.oracle.com/javase/7/docs/technotes/guides/javaws/developersguide/syntax.html#resources](https://docs.oracle.com/javase/7/docs/technotes/guides/javaws/developersguide/syntax.html#resources):

- -verbose
- -version
- -showversion
- -help
- -X
- -ea
- -enableassertions
- -da
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 runtime environment from the location specified in the JNLP file.

For more information about Web Start, see:
- [https://docs.oracle.com/javase/7/docs/technotes/guides/javaws/index.html](https://docs.oracle.com/javase/7/docs/technotes/guides/javaws/index.html)

For more information about deploying applications, see:
- [https://docs.oracle.com/javase/7/docs/technotes/guides/jweb/index.html](https://docs.oracle.com/javase/7/docs/technotes/guides/jweb/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.

**Important**: In line with Oracle changes, support is no longer available for Web Start from IBM SDK, Java Technology Edition, Version 7 Release 1. For continued support, move to the latest release of the IBM SDK for all applications launched or deployed through web browsers.

#### 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/java71[_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/java71[_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.

**Important:** In line with Oracle changes, support is no longer available for Web Start from IBM SDK, Java Technology Edition, Version 7 Release 1. For continued support, move to the latest release of the IBM SDK for all applications launched or deployed through web browsers.

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.

**Note:** The Classic plug-in, sometimes referred to as the First-generation plug-in, is deprecated in Java 7 and will not be supported in future releases. Instead, use the Next-generation plug-in to run applets in the browser.

**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/java71[_64]/` by `installp`. You can redistribute the SDK files in the `j7132redist.tar, j7164redist.tar` or `j7132redist.tar.gz, j7164redist.tar.gz` files (after viewing and agreeing to the associated online license) available from the AIX Java Web site: [https://www.ibm.com/developerworks/java/jdk/aix/](https://www.ibm.com/developerworks/java/jdk/aix/) Click the **Download and service information** link near the end of the page and follow the links to the Java download page.
Chapter 7. 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 by using the -version or -fullversion options. You can also obtain version information for all jar files on the class path by using the -Xjarversion option.
Procedure

1. Open a shell prompt.
2. Type the following command:
   ```
   java -version
   ```

   You will see information similar to the following:

   ```
   java version "1.7.0"
   Java(TM) SE Runtime Environment (build pxa6470_27-20130715_03(SR1))
   IBM J9 VM (build 2.7, JRE 1.7.0 Linux amd64-64 20130711_156087 (JIT enabled, AOT enabled)
   J9VM - R27_head_20130711_0404_B156087
   JIT - tr.r13.java_20130709_41534
   GC - R27_head_20130711_0404_B156087
   J9CL - 20130711_156087)
   JCL - 20130704_01 based on Oracle 7u25-b12
   ```

   The output provides the following information:
   - The first line indicates the Java standard edition class library level.
   - The second line includes information about the build level of the runtime environment. Service refresh (SR), fix pack (FP), and APAR (Interim fixes only) numbers are appended to the build string. In the example, the installed level is service refresh 1.
   - The third line indicates the build level of the Java virtual machine.
   - Subsequent lines provide detailed information about the levels of components that make up the runtime environment.

   Exact build dates and versions will change.
3. To obtain only the build information for the runtime environment, type the following command:
   ```
   java -fullversion
   ```

   You will see information similar to:

   ```
   java full version "JRE 1.7.0 IBM Windows 64 build
   pwa6470_27-20130715_03"
   ```

What to do next

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

   ```
   java -Xjarversion -version
   ```

   You will see information similar to:

   ```
   java version "1.7.0"
   Java(TM) SE Runtime Environment (build pxi3270_27-20130715_03)
   IBM J9 VM (build 2.7, JRE 1.7.0 Linux x86-32 20130711_156087 (JIT enabled, AOT enabled)
   J9VM - R27_head_20130711_0404_B156087
   JIT - tr.r13.java_20130709_41534
   GC - R27_head_20130711_0404_B156087
   J9CL - 20130711_156087)
   JCL - 20130704_01 based on Oracle 7u25-b12
   /opt/ibm/java-i386-70/jre/lib/ibmcfw.jar VERSION: CCX.CF [o1103.02]
   ...
The information available varies for each jar file and is taken from the 
*Implementation-Version* and *Build-Level* properties in the manifest of the jar file.

**Specifying Java options and system properties**

You can specify Java options and system properties directly on the command line. You can also use an options file or an environment variable.

**About this task**

The sequence of the Java options on the command line defines which options take precedence during startup. Rightmost options have precedence over leftmost options. In the following example, the *-Xjit* option takes precedence:

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

For more information about how the JVM constructs the runtime environment at startup, see “Specifying command-line options” on page 525.

Use one of more of the options that are shown in the procedure to customize your runtime environment.

**Procedure**

1. Specify options or system properties on the command line. For example:
   ```
   java -Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump MyJavaClass
   ```
2. Create an environment variable that is called *IBM_JAVA_OPTIONS* containing the options. For example:
   ```
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump"
   ```
3. Create a file that contains the options, and specify that file on the command line or in the *IBM_JAVA_OPTIONS* environment variable by using the *-Xoptionsfile* parameter. For more information about constructing this file, see “-Xoptionsfile” on page 558.

**Standard options**

The definitions for the standard options.

See “JVM command-line options” on page 549 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 Java runtime environments in the version search.

-no-jre-restrict-search
Exclude user private Java runtime environments 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 577 for more information) to control the output. SeeVerbose garbage collection logging for more information.

  jni
  Writes information to stderr describing the JNI services called by the application and JVM.

  sizes
  Writes information to stderr describing the active memory usage settings.

  stack
  Writes information to stderr describing the Java and C stack usage for each thread.

-version
Prints product version.

-version:<value>
Requires the specified version to run, for example “1.5”.

-X
Prints help on nonstandard options.

Globalization of the java command
The java and javaw launchers accept arguments and class names containing any character that is in the character set of the current locale. You can also specify any Unicode character in the class name and arguments by using Java escape sequences.

To do this, use the -Xargencoding command-line option.

-Xargencoding
Use argument encoding. To specify a Unicode character, use escape sequences in the form \\u####, where # is a hexadecimal digit (0 to 9, A to F).
-Xargencoding: utf8
  Use UTF8 encoding.

- Xargencoding: latin
  Use ISO8859_1 encoding.

For example, to specify a class called HelloWorld using Unicode encoding for both capital letters, use this command:
java -Xargencoding '\u0048ello\u0057orld'

The java and javaw commands provide translated messages. These messages differ based on the locale in which Java is running. The detailed error descriptions and other debug information that is returned by java is in English.

---

**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 install_dir/jre/bin/java
install_dir/jre/bin/java:
  ***Loader Section***
  Loader Header Information
  VERSION#   #SYMtableENT   #RELOCent   LENidSTR
  0x00000001  0x0000003f  0x0000006d  0x00000090
  #IMPfild   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 `lib_dir` and `lib_dir/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 `install_dir` 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 232.

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 in the AIX product documentation. For information about using either 64 KB or 16 GB pages with AIX, see Multiple page size support in the AIX product documentation. The default AIX page size is 4 KB.

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 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 232 for more information about possible settings. For example:

LDR_CNTRL=MAXDATA=0X80000000@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 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 236.

For more information about the JIT, see “JIT and AOT problem determination” on page 430.

### Disabling the JIT

The JIT can be disabled in a number of different ways. Both command-line options override the `JAVA_COMPILER` environment variable.

**About this task**

Turning off the JIT is a temporary measure that can help isolate problems when debugging Java applications.

**Procedure**

- Set the `JAVA_COMPILER` environment variable to `NONE` or the empty string before running the `java` application. Type the following command at a shell prompt:
  ```
  export JAVA_COMPILER=NONE
  ```
- Use the `-D` option on the JVM command line to set the `java.compiler` property to `NONE` or the empty string. Type the following command at a shell prompt:
  ```
  java -Djava.compiler=NONE <class>
  ```
- Use the `-Xint` option on the JVM command line. Type the following command at a shell prompt:
  ```
  java -Xint <class>
  ```
Enabling the JIT

The JIT is enabled by default. You can explicitly enable the JIT in a number of different ways. Both command-line options override the JAVA_COMPILER environment variable.

Procedure

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

  If the JAVA_COMPILER environment variable is an empty string, the JIT remains disabled. To disable the environment variable, at the prompt, enter:
  ```
  unset JAVA_COMPILER
  ```

- Use the -D option on the JVM command line to set the java.compiler property to jitc. At a prompt, enter:
  ```
  java -Djava.compiler=jitc <class>
  ```

- Use the -Xjit option on the JVM command line. Do not specify the -Xint option at the same time. At a prompt, enter:
  ```
  java -Xjit <class>
  ```

Determining whether the JIT is enabled

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

Procedure

Run the java launcher with the -version option. Enter the following command at a shell prompt:

```
java -version
```

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

```
(JIT disabled)
```

If the JIT is in use, a message is displayed that includes the following text:

```
(JIT enabled)
```

What to do next

For more information about the JIT, see The JIT compiler on page 105.

Specifying a garbage collection policy

The Garbage Collector manages the memory used by Java and by applications running in the JVM.

When the Garbage Collector receives a request for storage, unused memory in the heap is set aside in a process called “allocation”. The Garbage Collector also checks for areas of memory that are no longer referenced, and releases them for reuse. This is known as “collection”.

The collection phase can be triggered by a memory allocation fault, which occurs when no space remains for a storage request, or by an explicit System.gc() call.
Garbage collection can significantly affect application performance, so the IBM virtual machine provides various methods of optimizing the way garbage collection is carried out, potentially reducing the effect on your application.

For more detailed information about garbage collection, see “Detailed description of global garbage collection” on page 74.

**Garbage collection options**

The `-Xgcpolicy` options control the behavior of the Garbage Collector. They make trade-offs between throughput of the application and overall system, and the pause times that are caused by garbage collection.

The format of the option is as follows:

```
-Xgcpolicy:<value>
```

The following values are available:

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

- **balanced**
  
  The balanced policy uses mark, sweep, compact and generational style garbage collection. The concurrent mark phase is disabled; concurrent garbage collection technology is used, but not in the way that concurrent mark is implemented for other policies. The balanced policy uses a region-based layout for the Java heap. These regions are individually managed to reduce the maximum pause time on large heaps and increase the efficiency of garbage collection. The policy tries to avoid global collections by matching object allocation and survival rates. If you have problems with application pause times that are caused by global garbage collections, particularly compactions, this policy might improve application performance. If you are using large systems that have Non-Uniform Memory Architecture (NUMA) characteristics (x86 and POWER platforms only), the balanced policy might further improve application throughput. For more information about this policy, including when to use it, see “Balanced Garbage Collection policy” on page 84.

- **metronome**
  
  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. The metronome policy is supported on specific hardware and operating system configurations. For more information, see “Using the Metronome Garbage Collector” on page 218.

- **optavgpause**
  
  The "optimize for pause time" (optavgpause) policy uses concurrent mark and concurrent sweep phases. Pause times are shorter than with optthruput, but application throughput is reduced because some garbage collection work is taking place while the application is running. Consider using this policy if you have a large heap size (available on 64-bit platforms), because this policy limits...
the effect of increasing heap size on the length of the garbage collection pause. However, if your application uses many short-lived objects, the `gencon` policy might produce better performance.

**subpool**

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

**optthruput**

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

**More effective heap usage using compressed references**

Many Java application workloads depend on the Java heap size. The IBM SDK can use compressed references on 64-bit platforms to decrease the size of Java objects and make more effective use of the available space. The result is less frequent garbage collection and improved memory cache utilization.

If you specify the `-Xnocompressedrefs` command-line option, the 64-bit Java virtual machine (VM) stores object references as 64-bit values. If you specify the `-Xcompressedrefs` command-line option, object references are stored as 32-bit representation, which reduces the 64-bit object size to be the same as a 32-bit object.

As the 64-bit objects with compressed references are smaller than default 64-bit objects, they occupy a smaller memory footprint in the Java heap. This results in improved data locality, memory utilization, and performance. You might consider using compressed references if your application uses a lot of native memory and you want the VM to run in a small footprint.

Compressed references are used by default on a 64-bit IBM SDK when the value of `-Xmx`, which sets the maximum Java heap size, is in the correct range.

If you are using a 64-bit IBM SDK, you can use `-Xcompressedrefs` whenever you require a maximum heap size up to 25 GB. Larger heap sizes might result in an out of memory condition at run time because the VM requires some memory at low addresses. You might be able to resolve an out of memory condition in low addresses by using the `-Xmcrs` option.

See the `Compressed references` on page 73 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.

The -Xgcpolicy:metronome command-line option enables the metronome garbage
collection policy. This policy uses many short pauses, rather than a single longer
pause.

**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 \texttt{-Xgcppolicy:balanced} option requests a combined use of concurrent and partial garbage collection.

The \texttt{-Xgcppolicy:metronome} policy option uses a series of short pauses rather than a single long pause, and can provide shorter overall pause times. The metronome policy also accepts the command-line option \texttt{-Xgc:targetPauseTime=N}, which configures the maximum pause time, in milliseconds, for each individual garbage collection pause.

**Environments with very full heaps**

If the Java heap becomes nearly full, and very little garbage can be reclaimed, requests for new objects might not be satisfied quickly because no space is immediately available.

If the heap is operated at near-full capacity, application performance might suffer regardless of which garbage collection options are used; and, if requests for more heap space continue to be made, the application might receive an OutOfMemoryError, which results in JVM termination if the exception is not caught and handled. At this point, the JVM produces a Javadump file for use during diagnostic procedures. In these conditions, you are recommended either to increase the heap size by using the \texttt{-Xmx} option or to reduce the number of objects in use.

For more information, see "Garbage Collector diagnostic data" on page 436.

**Using the Metronome Garbage Collector**

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

AIX workload partitions (WP\textsc{ar}) and micropartitions are not supported. However, logical partitions (LP\textsc{ar}) are supported provided there is an integral number of processors. That is, LP\textsc{ars} with a fractional number of processors, such as 0.5 or 1.5, are not supported.

Support is available for both compressed and uncompressed references on 64-bit platforms. 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 73.

The Metronome Garbage Collector is a key component of the IBM WebSphere Real Time product. This 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](http://www-01.ibm.com/software/webservers/realtime/index.html)

**Configuring user accounts**

Important steps to configure AIX user accounts correctly on your system.
Procedure

This step must be completed one time only:

1. When the installation process is completed, you must change the user account to allow access to high-resolution timers. Run the following command as root user:
   
   `chuser "capabilities=CAP_NUMA_ATTACH,CAP_PROPAGATE" <username>`

   where `<username>` is the non-root AIX user account.

   **Note:** The user must log out and log back in for the change to take effect.

This step must be completed in every shell before starting Java:

1. Set the `AIXTHREAD_HRT` environment variable to true. This environment variable allows a process to use high-resolution timeouts with `clock_nanosleep()`. You must set this environment variable each time the process is started. On the command line, type:
   
   `AIXTHREAD_HRT=true`

   This setting can be added to a user's `.profile` so that it is set each time the user logs in. Add the following line to the user `.profile` file:

   `export AIXTHREAD_HRT=true`

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 - 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 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 577.

With the default target pause time set, the Verbose GC pause time graph shows that pause times are held around or under the 3 millisecond mark:
With a target pause time set at 6 milliseconds, the Verbose GC pause time graph shows that pause times are held around or under the 6 millisecond mark.
With a target pause time set at 10 milliseconds, the Verbose GC pause time graph shows that pause times are held around or under the 10 millisecond mark.

Figure 3. Actual pause times when the target pause time is set to 6 milliseconds
With a target pause time set at 15 milliseconds, the Verbose GC pause time graph shows that pause times are held around or under the 15 millisecond mark.

Figure 4. Actual pause times when the target pause time is set to 10 milliseconds
**Metronome Garbage Collector limitations**

Read about known limitations that you might encounter when you are using the metronome GC.

**Extended pause times**

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.

---

*Figure 5. Actual pause times when the target pause time is set to 15 milliseconds*
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.

**Running multiple JVMs**

Where possible, do not run more than one JVM that uses the metronome GC 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.

**Using shared classes cache**

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

A shared classes cache that is created for use with the metronome GC cannot be removed by a JVM that uses another GC policy.

---

**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.


To enable applications to respond to DLPAR events, the 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 API documentation for this package is available here: [API documentation](http://www.research.ibm.com/journal/sj42-1.html).

The launcher option, `-Xsoftmx`, is also available with the SDK. The `-Xmx` option specifies the maximum size (hard limit) of the Java heap. The `-Xsoftmx` option
specifies a smaller 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 to the value of `-Xsoftmx`. If unspecified, `-Xsoftmx` defaults to the value of `-Xmx`.

**Related information:**

[Virtualize this: Using Java API to exploit virtualization capabilities of the IBM POWER5 hardware](https://developerWorks.ibm.com/tech/virtualize-java-api-to-exploit-virtualization-capabilities-of-the-ibm-power5)

This developerWorks article describes the use of the java.lang.management API to respond to DLPAR events.

---

**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 225 for more information.
Using the IPv6 extensions

This release 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.

Support for bidirectional data

The runtime environment includes support for bidirectional data, where text is written from right to left and also from left to right.

Within the Java runtime environment, character data is manipulated as Unicode UTF-16 values. However, character data outside the Java runtime environment frequently conforms to different encodings. For this reason, file input/output operations, along with the conversion of bytes to characters and back, also involve conversion from an external encoding to UTF-16 and back. You can specify the external encoding, for example in the constructor of an `InputStreamReader` or `OutputStreamWriter` object, or rely on a default encoding.

Through its implementation of the Unicode standard, the Java runtime environment supports many languages, which have various alphabets or scripts. These languages include Arabic and Hebrew, whose scripts are written from right to left. Because Arabic and Hebrew text is frequently mixed with other languages (and numbers), which are written from left to right, the Java runtime environment must be able to handle bidirectional data.

Bidirectional data is more variable than non-bidirectional data because it can be stored not only in various encodings, but also in various layouts. Each layout is a combination of rules for the ordering of the characters (for Arabic and Hebrew) and the shaping of Arabic letters (choosing the appropriate shape of an Arabic letter from several possibilities).

For the same reasons that the Java runtime environment transforms data from external encodings into internal encodings and back, it should transform bidirectional data from external layouts to the layout used within the Java runtime environment, and back. For example, older applications might store data in a visual layout while Java APIs assume an implicit layout (also known as logical layout).

In the Java runtime environment, you can request that layout transformations are run for bidirectional data whenever encoding conversions are run. These transformations are disabled by default. To enable the transformations, set the `JAVABIDI` system property when you start your Java application. For more information, see "-Djavabidi" on page 538.

You can also use the Bidirectional Layout Engine API directly in your Java code. For more information, see bidirectional support in the API reference documentation.

Related information:

[Introduction to bidirectional languages]
Special Arabic characters

Some Arabic characters have different representations in different code pages, and so need special handling during code page conversion.

Because these characters are not represented in all code pages, a normal conversion results in substitute control characters (SUB), which is a loss of data.

Lam-Alef
This character is represented as a single character in code pages 420, 864, and 1046, which are used for visual presentation in addition to the Unicode Arabic Presentation Forms-B (uFExx range). This character is represented as two characters, Lam and Alef, in code pages 425, 1089, and 1256, which are used for implicit representation in addition to the Unicode Arabic u06xx range.

Tail of Seen family of characters
The visual code pages 420, 864, and 1046 represent the final form of the Seen family of characters as two adjacent characters: the three quarters shape and the Tail. The implicit code pages 425, 1089, 1256, and the Unicode Arabic u06xx range, do not represent the Tail character. In Unicode Arabic Presentation Forms-B (uFExx range), the final form for characters in the Seen family is represented as one character.

Tashkeel or diacritic characters except for Shadda
These characters are not represented in code pages 420 and 864. Conversion of Tashkeel characters from code pages 425, 1046, 1089, 1256, and Unicode to 420 or 864 results in SUB characters.

Yeh-Hamza final form
Code pages 420 and 864 have no unique character for the Yeh-Hamza final form; it is represented as two characters: Yeh final form and Hamza. In other code pages, such as 425, 1046, 1089, 1256, and Unicode, the Yeh-Hamza final form is represented as one character or two characters depending on the user's input; whether it is one key stroke (Yeh-Hamza key) or two strokes (Yeh key + Hamza key). The conversion from the previous code pages to 420 or 864 converts the Yeh-Hamza final form character to the Yeh-Hamza initial form; a special handling process must convert it to the Yeh final form and Hamza.

To avoid the loss of such characters during conversion, various Arabic shaping options are available to properly handle them.

Arabic shaping options
A set of shaping options is available for each Arabic character that requires special handling so that such characters are not lost during code page conversion.

Lam-Alef
During conversion from visual to implicit code pages, each Lam-Alef character is expanded to Lam plus Alef, consuming a blank space. If no blank space is available, the Lam-Alef character remains as it is in the Unicode uFExx range, and becomes a substitute control character (SUB) when it is converted to implicit single-byte code pages. During conversion from implicit to visual code pages, a space is generated by Lam-Alef compression. The position of the space that is consumed or generated depends on the shaping option, as described in the following table:
Table 12. The position of the blank space that is consumed or generated during code page conversion of the Lam-Alef character

<table>
<thead>
<tr>
<th>Shaping option</th>
<th>Position of the blank space that is consumed or generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Near</td>
<td>Next to the character that is being converted</td>
</tr>
<tr>
<td>At Begin</td>
<td>The beginning of the buffer (buffer[0])</td>
</tr>
<tr>
<td>At End</td>
<td>The end of the buffer (buffer[length - 1])</td>
</tr>
<tr>
<td>Auto</td>
<td>The beginning of the buffer relative to the orientation of the text: buffer[0] for left-to-right text and buffer[length - 1] for right-to-left text</td>
</tr>
</tbody>
</table>

**Seen Tail**

**Auto**

This option is deprecated, use the Near option instead, which has the same behavior.

**Near**

During conversion from visual to implicit code pages, each two-character final form of the Seen family of characters (comprising the three quarters shape character and the Tail character) is converted to the corresponding single-character final form, with a space replacing the Tail. The space is positioned next to the Seen character. During conversion from implicit to visual code pages, each single-character final form is converted to the corresponding two-character final form, consuming the space next to the Seen character. If no space is available, the character is converted to the single, three quarters shape character.

**Tashkeel**

**Auto**

No special processing is done.

**Customized At Begin**

All Tashkeel characters except for Shadda are replaced by spaces. The resulting spaces are moved to the beginning of the buffer (buffer[0]).

**Customized At End**

All Tashkeel characters except for Shadda are replaced by spaces. The resulting spaces are moved to the end of the buffer (buffer[length - 1]).

**Keep**

No special processing is done.

**Yeh-Hamza**

**Auto**

This option is deprecated, use the Near option instead, which has the same behavior.

**Near**

During conversion from visual to implicit code pages, each Yeh character that is followed by a Hamza character is converted to a Yeh-Hamza character. The space that results from the contraction process is positioned...
next to the Yeh-Hamza character. In conversion from implicit to visual code pages, each Yeh-Hamza character is expanded to two characters, Yeh and Hamza, consuming the space that is located next to the original Yeh-Hamza character. If no space is available, the Yeh-Hamza character is converted to the single character Yeh.

**Known limitations**

You might come across the following issues when you use bidirectional support.

- If an application program reads from a file, or writes to a file, pieces of text that do not constitute a logical unit, the bidirectional layout transformations produce unexpected results. For example, an application that reads or writes characters one at a time does not benefit from the bidirectional support. This limitation is not likely to be removed in future releases.

- When unmappable characters (characters that are not valid in the declared code page) appear in single-byte character set (SBCS) data, they might cause previous and following data to be transformed independently from one another, which can lead to unexpected results.

- When an application reads or writes a unit of text (a line for example) that crosses the boundary between buffers that are used by the input or output file, the bidirectional transformation might be done independently on the part of the text unit that is in each buffer, leading to unexpected results. When the file is not too large, you can avoid this problem by setting the buffer size to a value that is sufficient to contain the whole file. For example, you can specify the buffer size when you construct a BufferedInputStream or a BufferedOutputStream object.

---

**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, reduce the maximum native stack size using the `-Xss<size>` option.

The default native stack size is 1024 KB (64-bit SDK) or 512 KB (32-bit SDK). 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.

Related concepts:

"System resource limits and the `ulimit` command" on page 235

The operating system provides ways of limiting the amount of resource that can be used. Limits are set for each user, but are applied separately to each process that is running for that user. These limits can affect Java applications, for example if certain limits are too low, the system might not be able to generate a complete Java dump file.

---

### Configuring large page memory allocation

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

**About this task**

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

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

AIX 7.2

Large pages

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

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

For more information about the `-Xlp` options, see "JVM command-line options" on page 549.

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

The physical storage that is allocated for the object heap is allocated in increments of the page size. For example, if you use 2G large pages with the options `-Xmx1024M` and `-Xms512K`, the object heap is allocated on a 2G large page. The real memory for the 2G large page is allocated immediately. Even though the object heap is consuming a 2G large page in this example, the maximum object heap is still 1M with an initial Java heap of 512K, as specified.

For more information about the `-Xmx` option, see "Garbage Collector command-line options" on page 577.

Related concepts:
The operating system provides ways of limiting the amount of resource that can be used. Limits are set for each user, but are applied separately to each process that is running for that user. These limits can affect Java applications, for example if certain limits are too low, the system might not be able to generate a complete Java dump file.

**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](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_CNTRL=MAXDATA=0XB0000000@DSA` is set.
- Otherwise, `LDR_CNTRL=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 13. The default Java virtual memory model

<table>
<thead>
<tr>
<th>Segment</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AIX kernel</td>
</tr>
<tr>
<td>1</td>
<td>Java program</td>
</tr>
<tr>
<td>2</td>
<td>Primordial Stack (main program thread stack)</td>
</tr>
<tr>
<td>3</td>
<td>Native Heap (malloc’ed space)</td>
</tr>
<tr>
<td>4-C</td>
<td>Native Heap (malloc’ed space) or Memory Mapped space (mmap/shmat)</td>
</tr>
<tr>
<td>D</td>
<td>Shared library code</td>
</tr>
<tr>
<td>E</td>
<td>Memory Mapped space (mmap/shmat)</td>
</tr>
<tr>
<td>F</td>
<td>Shared library data</td>
</tr>
</tbody>
</table>

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 14. Memory model with MAXDATA=0xB0000000@DSA

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

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 15. Memory model with MAXDATA=0@DSA

<table>
<thead>
<tr>
<th>Segment</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AIX kernel</td>
</tr>
<tr>
<td>1</td>
<td>Java program</td>
</tr>
<tr>
<td>2</td>
<td>Native Heap (malloc'ed space) and Primordial Stack (main invocation stack)</td>
</tr>
<tr>
<td>3-F</td>
<td>Memory Mapped space (mmap/shmat)</td>
</tr>
</tbody>
</table>

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 16. Memory model with MAXDATA=0

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

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.

System resource limits and the ulimit command

The operating system provides ways of limiting the amount of resource that can be used. Limits are set for each user, but are applied separately to each process that is running for that user. These limits can affect Java applications, for example if certain limits are too low, the system might not be able to generate a complete Java dump file.

Limits can be hard or soft. Hard limits are set by the root user. Only the root user can increase hard limits, though other users can decrease them. Soft limits can be set and changed by other users, but they cannot exceed the hard limits. To view the current limits, enter the following command:

`ulimit -Xa`

where X is either H for hard limits, or S for soft limits. If you do not specify a value for X, the soft limits are displayed.

Setting temporary limits

Use the `ulimit` command to set limits that apply only to your current session. If you use the command in a script, the settings apply during the run time of the script. The settings also apply to processes that are created by the script, or other scripts that are called from within the first script. The format of the command is as follows:

`ulimit -[H|S]limit_name limit_value`

Where H indicates a hard limit, and S indicates a soft limit. If you omit this parameter, the soft limit is set. The following example sets a soft file size limit to unlimited:

`ulimit -f unlimited`
Storing limit settings

Use the /etc/security/limits file to store ulimit settings. If you are setting a hard and a soft limit, set the hard limit first in the file. Settings can be default, or specific to individual users or groups. Changes to this file should be made by a system administrator.

Note: Different work partitions might have different limit files.

View the documentation for your version of the operating system for instructions on how to edit the file, because the steps can vary between versions. The file itself might contain instructions in the commented section.

Changes to the file take effect when you start a new login shell, for example bash -l, or if you log out and then log back in to the system.

Note: The init command, and its child processes, do not use the limits that are specified in the limit file. When the operating system starts, the init command runs the commands that are listed in the inittab file, or in scripts in the init.d directory. These actions create child processes of the init command. At this point, the initialization process has not yet loaded the settings in the limit file. These settings are loaded only when a login terminal session is started, or when a user logs in to the system. If you add a command to the inittab file, or to scripts in the init.d directory, and you want the command to use specific user limits, you must include actions to set the user limit values as required.

Available limits

The limits that you can set vary between versions of the operating system; view the documentation for your version for details.

Related information:

AIX information centers

View more detailed information about user limits, for your version of AIX.

“Enabling full AIX core files” on page 256

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

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 [Stack Execution Disable protection](#) in the AIX product documentation.
Chapter 8. Performance

You can improve the performance of applications by tuning the product, enabling hardware features, or using specific APIs in your application code.

Here are some methods for improving performance:

- Share class data between VMs. For more information, see "Class data sharing between JVMs."

- Choose the best garbage collection policy for your application. For more information, see "Specifying a garbage collection policy" on page 214.

- Enable large page support on your operating system. For more information, see "Configuring large page memory allocation" on page 231.

- Enable hardware compression acceleration, available on some systems. For more information, see "Enabling hardware compression acceleration on Power Systems" on page 246.

- If your code interacts with objects that are not in Java code, use packed objects to optimize the interactions. For more information, see "Packed objects" on page 16.

- Use the Data Access Accelerator API in your Java code

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 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 246.

The JVM can also store ahead-of-time (AOT) compiled code in the cache for certain methods to improve the startup time of subsequent JVMs. The AOT compiled code is not shared between JVMs, but is cached to reduce compilation time when the JVM starts. The amount of AOT code stored in the cache is determined heuristically. You cannot control which methods get stored in the cache. You can set maximum and minimum limits on the amount of cache space used for AOT code, or you can disable AOT caching completely. See “Class data sharing command-line options” on page 241 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 241 for more information.

Cache access

A JVM can access a cache with either read/write or read-only access. Any JVM connected to a cache with read/write access can update the cache. Any number of JVMs can concurrently read from the cache, even while another JVM is writing to it.

You must take care if runtime bytecode modification is being used. See “Runtime bytecode modification” on page 244 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 read/write access by default unless the groupAccess command-line suboption is used, in which case the access is read/write for user and groups.

Service refresh 2 and later: When a process attempts to access a shared class cache, the virtual machine grants or denies access based on the user ID of the process and the creator of the cache as follows:

- Access is granted to the user that created the cache.
- Access is granted to any other user that is in the same group as the cache creator, but only if the -Xshareclasses:groupAccess option is specified on the command line.
- Access is denied in all other cases. For example, even if the cache has read permission for all, access is denied unless one of the previous points also applies.
Note: These checks are not run for shared cache utility options such as
-Xshareclasses:printStats, -Xshareclasses:destroy, or
-Xshareclasses:destroyAll.

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 245 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<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” or use -Xshareclasses:help to see a list of
available suboptions.

Related reference:
-Xshareclasses” on page 560
Enables class sharing. This option can take a number of suboptions, some of which
are cache utilities.
-Xscmx” on page 560
Specifies the size of a new cache.

Class data sharing command-line options

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

For more information about an option, click the link. 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.

-Xshareclasses” on page 560
Enables class sharing. This option can take a number of suboptions, some
of which are cache utilities.
The following options can be used on the command line with the \texttt{-Xshareclasses} option:

\texttt{-Xscmaxaot} on page 577
Applies a maximum number of bytes in the class cache that can be used for AOT data.

\texttt{-Xscmaxjitdata} on page 577
Applies a maximum number of bytes in the class cache that can be used for JIT data.

\texttt{-Xscminaot} on page 577
Applies a minimum number of bytes in the class cache to reserve for AOT data.

\texttt{-Xscminjitdata} on page 577
Applies a minimum number of bytes in the class cache to reserve for JIT data.

\texttt{-Xscdmx} on page 559
Controls the size of the class debug area when you create a shared class cache.

\texttt{-Xscmx} on page 560
Specifies the size of a new shared class cache.

\texttt{-Xzero} on page 568
Enables reduction of the memory footprint of the Java runtime environment when concurrently running multiple Java invocations. This option is deprecated in IBM SDK, Java Technology Edition, Version 7 Release 1, and will be removed from a future release.

\textbf{Creating, populating, monitoring, and deleting a cache}

An overview of the lifecycle of a shared class data cache, including examples of the cache management utilities.

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

The Java virtual machine (VM) 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 that are being loaded until the cache becomes full. If two or more VMs are started concurrently, they populate the cache concurrently.

To check that the cache is created, run \texttt{java -Xshareclasses:listAllCaches}. To see how many classes and how much class data is being shared, run \texttt{java -Xshareclasses:[name=<name>],printStats}. You can run these utilities after the application VM ends or in another command window.

For more feedback on cache usage while the VM is running, use the \texttt{verbose} suboption. For example, \texttt{java -Xshareclasses:[name=<name>],verbose}.

To see classes that are being loaded from the cache or stored in the cache, add \texttt{-Xshareclasses:[name=<name>],verboseIO} to your command line when you run your application.

Caches can be deleted if they contain many stale classes or if the cache is full and you want to create a bigger cache. To delete a cache, run \texttt{java}
-Xshareclasses:[name=<name>],destroy. If you want to delete a 64-bit non-compressed references cache, run java -Xshareclasses:[name=<name>],destroy -Xnocompressedrefs.

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, by using the -Xscmx option. Then, run the application and use the printStats option to determine how much class data is stored. Add a small amount to the value shown in the printStats output for contingency. Because classes can be loaded at any time during the lifetime of the VM, it is best to do this analysis after the application ends. However, a full cache does not have a negative affect on the performance or capability of any VMs connected to it. Therefore, you can choose a cache size that is smaller than required.

If a cache becomes full, a message is displayed on the command line of any VMs that are using the verbose suboption. All VMs sharing the full cache can then load 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.

Related reference:
- “-Xshareclasses” on page 560
  Enables class sharing. This option can take a number of suboptions, some of which are cache utilities.
- “-Xscmx” on page 560
  Specifies the size of a new cache.

Performance and memory consumption

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

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

Duplicate classes are consolidated in the shared class cache. For example, class A loaded from myClasses.jar and class A loaded from myOtherClasses.jar (with identical content) is stored only once in the cache. The printAllStats utility shows multiple entries for duplicated classes, with each entry pointing to the same class.

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

Considerations and limitations of using class data sharing

Consider these factors when deploying class data sharing in a product and using class data sharing in a development environment.
Cache size limits
The maximum theoretical cache size is 2 GB. The size of cache you can specify is limited by the amount of physical memory and paging space available to the system.

The shared class cache consists of memory mapped files that are created on disk and remain when the operating system is restarted. If you change the default behavior using the -Xshareclasses:nonpersistent option, so that the cache is not retained on restart, the cache for sharing classes is allocated using the System V IPC shared memory mechanism.

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 232.

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

Related reference:
«-Xshareclasses” on page 560
Enables class sharing. This option can take a number of suboptions, some of which are cache utilities.

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.

The option **-Xshareclasses:enableBCI** is now enabled by default. If you are migrating from IBM SDK, Java Technology Edition, Version 7, or earlier releases, you must set **disableBCI** when using the **modified=<modification_context>** suboption to retain the same behavior. For more information see, “Class data sharing command-line options” on page 241.

See “Dealing with runtime bytecode modification” on page 471 for more detail on this topic.

**Operating system limitations**

You cannot share classes between 32-bit and 64-bit Java virtual machines (VM). 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 allowed between 32-bit and 64-bit VMs. The **listAllCaches** suboption lists 32-bit and 64-bit caches, depending on the address mode and compressed references mode of the VM 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 VM cannot identify the shared classes on the system and must re-create the cache. Use the **ipcs** command to view the memory segments that are used by a VM or application.

Users running a Java VM 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.

**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 245.

IBM provides several Java interfaces for various types of custom class loaders, which allow the class loaders to find and store classes in the shared class cache. These classes are in the com.ibm.oti.shared package.

The API documentation for this package is available here: API documentation

See “Using the Java Helper API” on page 477 for more information about how to use these interfaces.

Enabling hardware compression acceleration on Power Systems

You can use hardware compression acceleration to increase the speed of data compression and decompression on some AIX Power Systems. If your application uses java.util.zip classes to provide Java compression services, hardware compression can reduce CPU consumption and shorten processing times.

Before you begin

Hardware compression is done by a PCIe data compression/decompression card, driven by the Generic Work Queue Engine (GenWQE) device driver that is provided in some operating systems. If you want your Java applications to use hardware compression services, your system must meet the following software and hardware requirements:

• One of the following operating systems:
  – AIX 6.1, Technology Level 9, or later
  – AIX 7.1, Technology Level 3, Service Pack 2, or later
• GenWQE device driver. The aforementioned operating systems already include this driver.
• PCIe3 LP Field Programmable Gate Array (FPGA) Accelerator Adapter

About this task

Accelerated Data Compression is provided as part of the IBM developer kit Zip native library, which you call by using the java.util.zip application programming interface.

Procedure

1. Set the following environment variables:
   
   ```bash
   export ZLIB_DEFLATE_IMPL=1
   export ZLIB_INFLATE_IMPL=1
   ```

   These environment variables specify whether hardware or software compression or decompression is used. By default, both variables are set to 0, which specifies that software compression services are used for both compression and decompression for Java applications.

2. Check that the input buffers for your Java application are sufficiently large.
Some CPU resource is required to send data to the hardware compression accelerator. For small amounts of data, this resource cost can be greater than the savings that are achieved by using hardware compression services. You can set a threshold value for the data that is to be sent to hardware compression services, by using the ZLIB_INFLATE_THRESHOLD variable. If the size of the data is below the threshold (default 16 KB), software compression is used instead. Here are some methods for determining an application’s current buffer size:

- Examine the source code
- Examine the behavior of the application by tracing it (see "Tracing Java applications and the JVM" on page 393), or monitoring it with IBM Monitoring and Diagnostic Tools

3. In your Java application, use the classes and methods in the java.util.zip package to compress and decompress data. The following example (which excludes imports and try/catch blocks) uses the GZIPOutputStream class to read data from one file and write compressed data to another file:

```java
// This 64 KB input buffer exceeds the threshold value set by ZLIB_INFLATE_THRESHOLD,
// so is eligible for hardware compression:
byte buffer[] = new byte[64 * 1024];
byte outputFile[];

input = new FileInputStream(argv[0]);
output = new ByteArrayOutputStream();
gzStream = new GZIPOutputStream(output, 4096);

for(;;) {
    // Read data from an uncompressed file:
    readBytes = input.read(buffer);
    if(readBytes < 0) {
        break;
    }
    else { // Write data to a compressed file:
        gzStream.write(buffer, 0, readBytes);
    }
}
```

**Results**

If your system meets the requirements and conditions that are described, accelerated hardware data compression is used. Otherwise, software-based compression services are used for data compression and decompression.

If you experience problems, see “Hardware compression acceleration issues on Power Systems” on page 307.

**Related information:**

- developerWorks article: Accelerated Data Compressing using the GenWQE Linux Driver and Corsa FPGA PCIe card
- zEnterprise Data Compression (zEDC)

**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 283.
Chapter 9. Security

The product contains components and tools that you can use to increase the security of your Java applications, for example by creating keys or using APIs such as Java Generic Security Services (JGSS). Some other features of the product, for example shared classes, also have security aspects that you should be aware of. Read the subtopics for an overview; more detailed information is provided in the rest of the user guide or in the security guide.

If a security vulnerability is found in the IBM or Oracle code, it is documented in the Security alerts page in the JavaSDK developer center, and a fix provided in the Fixes in the IBM Developer Kits page. If you want to receive security bulletins and fix notifications, subscribe to the My Notifications service, selecting the appropriate document types. You can also choose to receive notifications about other document types such as news and flash alerts. Security bulletins for all IBM products are published on the IBM Product Security Incident Response blog site.

Security considerations

Review the following information so that you are aware of the security issues and solutions that are available in the product. Some product features affect the security of the runtime environment itself, others affect the security of applications that you run in the environment:

Attach API
You can use the Java Attach API to connect an application to a different virtual machine. Security is handled by POSIX file permissions. Check and secure access to ensure that only authorized users or processes can connect to another virtual machine, or disable the Java Attach API capability by specifying a system property. For more information, see “Support for the Java Attach API” on page 195.

Dump files
Be careful when handling dump files, because they can contain all the information from your application, some of which might be sensitive. For example, dump files can contain personal information or bank account details. For more information about dump files, see “Using diagnostic tools” on page 312.

JConsole
JConsole is a graphical tool which you can use to monitor and manage the behavior of Java applications. You can specify options to disable password authentication and encryption, allowing any JMX agent to connect to your Java application. Use these non-secure options only in a development or testing environment. For more information, see “Using JConsole” on page 520.

Hashing algorithms
Enhanced, more secure hashing algorithms are available for the following APIs. Enhanced, more secure hashing algorithms are available for the following APIs. These algorithms can change the iteration order of items returned from hashed maps. You can use system properties to enable or disable these algorithms, if required for compatibility:
• java.util.HashMap, java.util.Hashtable, java.util.LinkedHashMap, java.util.WeakHashMap, and java.util.concurrent.ConcurrentHashMap. The enhanced algorithm for these APIs is disabled by default. For more information, see “-Djdk.map.althashing.threshold” on page 541.

• javax.xml.namespace.QName.hashCode(). The enhanced algorithm for this API is enabled by default. For more information, see “-Djavax.xml.namespace.QName.useCompatibleHashCodeAlgorithm” on page 541.

Securing XML processing
If your application takes untrusted XML, XSD or XSL files as input, you can enforce specific limits during Java API for XML (JAXP) processing to protect your application from malformed data. For more information, see “Securing Java API for XML processing (JAXP) against malformed input” on page 161.

Shared classes
You can share class data between virtual machines by storing it in a cache, which can reduce virtual storage consumption and startup time for virtual machines.

• Access to the shared class cache is limited by operating system permissions and Java security permissions. For more information, see the Cache security section of the following topic: “Overview of class data sharing” on page 239.

• You can also restrict access to the cache by specifying the cache location, the permissions for that location, and by including user names in cache names. For more information, see Cache naming.

Secure static versioning
Static versioning allows Web Start applications to request a specific VM version on which those applications will run. Applications can therefore use this feature to exploit old security vulnerabilities on systems that were upgraded to a new VM. Use Secure Static Versioning (SSV) instead. For more information, see: “WebStart Secure Static Versioning” on page 206.

Note: In line with Oracle changes, this feature is not available from service refresh 3, fix pack 50.

Java security
By default, Java security is not enabled, which allows applets or untrusted code to run without restrictions. For example, read or write to a file, read process memory, or start new processes. In order to secure Java the SecurityManager needs to be enabled, which can be achieved by specifying the -Djava.security.manager system property on the command line when you start your application.

Security settings for the Java Plug-in and Web Start, in the Java Control Panel
You use the Java Control Panel to control how Java applications that are embedded in a browser, or launched from a browser, run on your computer. Some of the settings in the Java Control Panel affect security, for example you can prevent all Java applications from running in, or from, a browser. For more information, see [Java Control Panel] but note that there are some differences in the IBM implementation of the control panel. For example, there is no Update tab.

Security components
The SDK provides security components that contain APIs and tools for securing your Java applications. These components cover areas such as
cryptography, keys and certification, access control, secure communication, and authentication, by using standards such as the following:

- Java Certification Path (CertPath)
- Java Authentication and Authorization Service (JAAS)
- Java Cryptography Extension (JCE)
- Java Generic Security Services (JGSS)
- Java Secure Socket Extension 2 (JSSE2)
- PKCS 11
- Simple Authentication and Security Layer (SASL)
- Java XML Digital Signature and Encryption

For more information, see “Security components and tools.”

**Upgrading**

An SDK upgrade can overwrite configuration files and security policy files. Back up these files in case you need to restore them after the upgrade. For more information, see “Upgrading the SDK” on page 153.

**Other**

The following topics might also contain information about security:

- “Known issues and limitations” on page 606
- “What’s new” on page 6
- “Migrating from earlier releases of the IBM SDK, Java Technology Edition” on page 143

The IBM SDK is based on Java Technology developed by Oracle Corporation, so also refer to the documentation available on the Oracle website. For example:

- Secure Coding Guidelines for Java SE
- Security Manager API
- RMI security considerations

**Security components and tools**

The security components and utilities that are listed here are shipped with IBM SDK, Java Technology Edition, Version 7 Release 1. The security components contain the IBM implementation of various security algorithms and mechanisms.

The following list summarizes the IBM security components and utilities that are available with the SDK. Further information about IBM security, including samples and API documentation, can be found here: Security Guide for IBM SDK, Java Technology Edition.

- Java Certification path
- Java Authentication and Authorization Service (JAAS)
- Java Cryptographic Extension (JCE)
- Java Cryptographic Extension (JCE) FIPS
- IBM SecureRandom provider
- Java Generic Security Services (JGSS)
- Java Secure Socket Extension 2 (JSSE2)
- Public Key Cryptographic Standard (PKCS #11) Implementation Provider
- Simple Authentication and Security Layer (SASL)
- IBM Key Certificate Management
- Java XML Digital Signature
- Java XML Encryption
- IBM Common Access Card (CAC) provider
- iKeyman
- Keytool

By default, the IBM® SDK, on all platforms, provides strong but limited JCE jurisdiction policy files. If you want to use unlimited jurisdiction policy files, you must obtain these files from Oracle.
Chapter 10. 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 the IBM SDK, make a report through the product that supplied the SDK, or through the operating system if there is no bundling product.

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”
• “AIX problem determination” on page 255
• “ORB problem determination” on page 293
• “NLS problem determination” on page 291

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 here: [Java SDK developer center](https://www.ibm.com/developerWorks/java/)

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 253 section, and check for latest details on the Developer Works website [https://www.ibm.com/developerWorks/](https://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 430.)

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 321 for details. The operating system settings might also need to be in place to enable the dump to be generated and to ensure that it is complete. Details of the required operating system settings are contained in the relevant problem determination section for the platform.

What logging information is available?

The JVM logs information about problems as they occur. You can enable more detailed logging, and control where the logging information goes. For more details, see “JVM messages” on page 595.

AIX problem determination

This section describes problem determination on AIX.

The topics are:

- “Setting up and checking your AIX environment” on page 257
- “General debugging techniques” on page 257
- “Diagnosing crashes” on page 269
- “Debugging hangs” on page 270
- “Understanding memory usage” on page 273
- “Debugging performance problems” on page 283
- “MustGather information for AIX” on page 290

Setting up and checking your AIX environment

Set up the correct 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 Java runtime 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 /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 installation by using smitty or installp. You can find the list of prerequisites in the User Guide that is shipped with the SDK. Use lslpp -1 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 321 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 for the user running the JVM:

   - ulimit -c unlimited to turn on corefiles with unlimited size
   - ulimit -n unlimited to allow an unlimited number of open file descriptors
   - ulimit -d unlimited to set the user data limit to unlimited
   - ulimit -f unlimited to set the file limit to unlimited

   For more information about ulimit settings, see "System resource limits and the ulimit command" on page 235.

   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 options 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:

cdev -l sys0 -a fullcore='true' -a430core='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 command:

cjava -Xdump:what

which should produce something like the following output:

-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 321.

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.

Maximum file size

The target file system for the system dump must support file sizes that are as large as the system dump, as well as an appropriate ulimit. If the supported file size is too small, dumps might be truncated to the maximum supported file size. To check your current file size and learn how to create a filesystem with a file size greater than 2GB, see https://www.ibm.com/support/docview.wss?uid=isg3T1023245

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.

You can get further information from the AIX documentation in IBM Knowledge Center. Of particular interest are (links to AIX version 7.1 information):

• 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](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 340.
- Starting Heapdumps, see “Using Heapdump” on page 363.
- Starting system dumps, see “Using system dumps and the dump viewer” on page 373.

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 SDK also includes a dbx Plug-in for additional help debugging Java applications. See “DBX Plug-in” on page 268 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:

```bash
# 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 260 for an example.

**netpmon**
- uses the trace facility to obtain a detailed picture of network activity during a time interval. See “netpmon” on page 261 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 262 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 https://www.ibm.com/developerworks/mydeveloperworks/wikis/home/wiki/Power%20Systems/page/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 263 for more details.

sar
Shows usage by multiple CPUs. See “sar” on page 264 for more details.

svmon
Captures snapshots of virtual memory. See “svmon” on page 265 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 266 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 267 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 267 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: 3HO41021
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

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
* vio0 Asynchronous Terminal
* pci12 PCI Bus
* pci11 PCI Bus
* pci10 PCI Bus
* pci9 PCI Bus
* pci8 PCI Bus
+ scsi0 Wide/Ultra-3 SCSI I/O Controller
+ hdi nk0 16 Bit LVD SCSI Disk Drive (73400 MB)
+ ses0 SCSI Enclosure Services Device
* pci7 PCI Bus
* pci6 PCI Bus
* pci5 PCI Bus
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:**

```
-------------------------------
| Process (top 20) | PID | CPU Time | CPU % | Network |
-------------------------------
| 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:

```
Device | Xmit | Recv |
-------|------|------|
| 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.
```
netpmom -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:
```
netpmom -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:
```
Usage of in: 
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:

<table>
<thead>
<tr>
<th>UID</th>
<th>PID</th>
<th>PPID</th>
<th>C</th>
<th>STIME</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>user12</td>
<td>29730</td>
<td>27936</td>
<td>0</td>
<td>21 Jun</td>
<td>-</td>
<td>12:26</td>
<td>java StartCruise</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>USER</th>
<th>PID</th>
<th>PPID</th>
<th>TID</th>
<th>ST</th>
<th>CP</th>
<th>PRI</th>
<th>SC</th>
<th>WCHAN</th>
<th>F</th>
<th>2000001</th>
<th>pts/10</th>
<th>TT</th>
<th>BND</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>user12</td>
<td>29730</td>
<td>27936</td>
<td>-</td>
<td>A</td>
<td>4</td>
<td>60</td>
<td>8</td>
<td>*</td>
<td>200001</td>
<td>pts/10</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>java StartCruise</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>31823</td>
<td>S</td>
<td>0</td>
<td>60</td>
<td>1</td>
<td>e6007cbc</td>
<td>8400400</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>44183</td>
<td>S</td>
<td>0</td>
<td>60</td>
<td>1</td>
<td>e6009acbc</td>
<td>8400400</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>83405</td>
<td>S</td>
<td>2</td>
<td>60</td>
<td>1</td>
<td>50c72558</td>
<td>400400</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>114071</td>
<td>S</td>
<td>0</td>
<td>60</td>
<td>1</td>
<td>e6011bdbc</td>
<td>8400400</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>116243</td>
<td>S</td>
<td>2</td>
<td>61</td>
<td>1</td>
<td>e601cb6bc</td>
<td>8400400</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
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
```

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:

<table>
<thead>
<tr>
<th>Pid</th>
<th>Command</th>
<th>Inuse</th>
<th>Pin</th>
<th>Pgsp</th>
<th>Virtual</th>
<th>64-bit</th>
<th>Mthrd</th>
</tr>
</thead>
<tbody>
<tr>
<td>25804</td>
<td>App5</td>
<td>78907</td>
<td>1570</td>
<td>182</td>
<td>67840</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Vsid</td>
<td>Esid</td>
<td>Type</td>
<td>Description</td>
<td>Inuse</td>
<td>Pin</td>
<td>Pgsp</td>
<td>Virtual</td>
</tr>
<tr>
<td>2c7ea</td>
<td>3</td>
<td>work</td>
<td>shmat/mmap</td>
<td>36678</td>
<td>0</td>
<td>0</td>
<td>36656</td>
</tr>
<tr>
<td>3c80e</td>
<td>4</td>
<td>work</td>
<td>shmat/mmap</td>
<td>7956</td>
<td>0</td>
<td>0</td>
<td>7956</td>
</tr>
<tr>
<td>5cd36</td>
<td>5</td>
<td>work</td>
<td>shmat/mmap</td>
<td>7946</td>
<td>0</td>
<td>0</td>
<td>7946</td>
</tr>
<tr>
<td>14e04</td>
<td>6</td>
<td>work</td>
<td>shmat/mmap</td>
<td>7151</td>
<td>0</td>
<td>0</td>
<td>7151</td>
</tr>
<tr>
<td>7001c</td>
<td>d</td>
<td>work</td>
<td>shared</td>
<td>6781</td>
<td>0</td>
<td>0</td>
<td>736</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>work</td>
<td>kernel</td>
<td>4218</td>
<td>1552</td>
<td>182</td>
<td>3602</td>
</tr>
<tr>
<td>6cb5a</td>
<td>7</td>
<td>work</td>
<td>shmat/mmap</td>
<td>2157</td>
<td>0</td>
<td>0</td>
<td>2157</td>
</tr>
<tr>
<td>48733</td>
<td>c</td>
<td>work</td>
<td>shmat/mmap</td>
<td>1244</td>
<td>0</td>
<td>0</td>
<td>1244</td>
</tr>
<tr>
<td>cac3</td>
<td>-</td>
<td>pers</td>
<td>/dev/hd2:176297</td>
<td>1159</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>54bb5</td>
<td>-</td>
<td>pers</td>
<td>/dev/hd2:176307</td>
<td>473</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>78b9e</td>
<td>-</td>
<td>pers</td>
<td>/dev/hd2:176301</td>
<td>454</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>58b66</td>
<td>-</td>
<td>pers</td>
<td>/dev/hd2:176308</td>
<td>254</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>cee2</td>
<td>-</td>
<td>work</td>
<td></td>
<td>246</td>
<td>17</td>
<td>0</td>
<td>246</td>
</tr>
<tr>
<td>4cbb3</td>
<td>-</td>
<td>pers</td>
<td>/dev/hd2:176305</td>
<td>226</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7881e</td>
<td>-</td>
<td>pers</td>
<td>/dev/e2axa702-1:1:2048</td>
<td>186</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>68f5b</td>
<td>-</td>
<td>pers</td>
<td>/dev/e2axa702-1:2048</td>
<td>185</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>288a</td>
<td>-</td>
<td>pers</td>
<td>/dev/hd2:176299</td>
<td>119</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>108c4</td>
<td>-</td>
<td>pers</td>
<td>/dev/e2axa702-1:1843</td>
<td>109</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>24b68</td>
<td>f</td>
<td>work</td>
<td>shared</td>
<td>97</td>
<td>0</td>
<td>0</td>
<td>78</td>
</tr>
<tr>
<td>64bb9</td>
<td>-</td>
<td>pers</td>
<td>/dev/hd2:176311</td>
<td>93</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>74bbd</td>
<td>-</td>
<td>pers</td>
<td>/dev/hd2:176315</td>
<td>68</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3082d</td>
<td>2</td>
<td>work</td>
<td>process</td>
<td>68</td>
<td>1</td>
<td>0</td>
<td>68</td>
</tr>
<tr>
<td>10bc4</td>
<td>-</td>
<td>pers</td>
<td>/dev/hd2:176322</td>
<td>63</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>50815</td>
<td>1</td>
<td>pers</td>
<td>code,/dev/hd2:210969</td>
<td>9</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>44bb1</td>
<td>-</td>
<td>pers</td>
<td>/dev/hd2:176303</td>
<td>7</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7c83e</td>
<td>-</td>
<td>pers</td>
<td>/dev/e2axa702-1:2048</td>
<td>4</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>34a6c</td>
<td>a</td>
<td>mmap</td>
<td>mapped to sid 44ab0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>70b3d</td>
<td>8</td>
<td>mmap</td>
<td>mapped to sid 1c866</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5cb36</td>
<td>b</td>
<td>mmap</td>
<td>mapped to sid 7cb5e</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5bb37</td>
<td>9</td>
<td>mmap</td>
<td>mapped to sid 1cb66</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1c7c7</td>
<td>-</td>
<td>pers</td>
<td>/dev/hd2:243801</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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
Mon Apr 16 16:16:50 2001 Interval: 2

Kernel 63.1
User 36.8
Wait 0.0

EVENTS/QUEUES
Cswitch 5984
Readch 4864
Syscall 15776

FILE/TTY
Writest 34280
Reads 8
Rawin 0
Writech 498

Forks 0
Igets 0
### 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:

```bash
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 increase, especially beyond 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. The stack size might be 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 the first in 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 remaining, 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 VM 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 382](#).
- 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.
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 430].

### 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 258]. 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 350].
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 351. 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
```

<table>
<thead>
<tr>
<th>USER</th>
<th>PID</th>
<th>PPID</th>
<th>TID ST</th>
<th>CP PRI SC</th>
<th>WCHAN</th>
<th>F</th>
<th>TT BND</th>
<th>COMMAND</th>
</tr>
</thead>
</table>
| wuser | 43824 | 51762 | -A 66 60 77 | * 200001 | pts/4 | - | java ...

... - - - 4021 $ 0 60 1 22c4d670 c00400 - - -
- - - 11343 $ 0 60 1 e6002cbc 8400400 - - -
- - - 14289 $ 0 60 1 22c4d670 c00400 - - -
- - - 14379 $ 0 60 1 22c4d670 c00400 - - -

... - - - 43187 $ 0 60 1 701e6114 400400 - - -
- - - 43939 R 33 76 1 20039c88 c00000 - - -
- - - 50275 $ 0 60 1 22c4d670 c00400 - - -
- - - 52477 $ 0 60 1 e600ccbc 8400400 - - -

... - - - 98911 $ 0 60 1 7023d46c 400400 - - -
- - - 99345 R 33 76 0 - 400000 - - -
- - - 99877 $ 0 60 1 22c4d670 c00400 - - -
- - - 100661 $ 0 60 1 22c4d670 c00400 - - -
- - - 102599 $ 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
$1 wait 0xe60196bc blocked 104099 k no sys _pthread_mutex_lock
> $2 wait 0xe60196bc blocked 104099 k no sys _pthread_mutex_lock
$3 wait 0x2015a458 running 29871 k no sys _pthread_mutex_lock
...
$50 wait running 86077 k no sys getLinkRegister
$51 run running 43939 u no sys reverseHandle
$52 wait running 56273 k no sys getLinkRegister
$53 wait running 37797 k no sys getLinkRegister
$60 wait running 4021 k no sys getLinkRegister
```

Chapter 10. Troubleshooting and support  271
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
$tx1  run running 18791  k  no  sys reverseHandle
$tx2  run running 99345  k  no  sys reverseHandle
$tx3  run running 20995  k  no  sys reverseHandle
```

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
$tx1  run running 43939  u  no  sys reverseHandle
```

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)
java.io.PrintWriter.flush(Compiled Code)
java.io.PrintWriter.flush(Compiled Code)
java.io.PrintWriter.flush(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 283 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-bit 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-bit 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.
### 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.

---

**Table: Memory Segments**

<table>
<thead>
<tr>
<th>Segment</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>0x0</td>
</tr>
<tr>
<td>Application program text</td>
<td>0x1</td>
</tr>
<tr>
<td>Application program data and application stack</td>
<td>0x2</td>
</tr>
<tr>
<td>Shared memory and mmap services</td>
<td>0x3</td>
</tr>
<tr>
<td></td>
<td>0x4</td>
</tr>
<tr>
<td></td>
<td>0x5</td>
</tr>
<tr>
<td></td>
<td>0x6</td>
</tr>
<tr>
<td>Shared library text</td>
<td>0x7</td>
</tr>
<tr>
<td>Miscellaneous kernel data</td>
<td>0x8</td>
</tr>
<tr>
<td></td>
<td>0x9</td>
</tr>
<tr>
<td></td>
<td>0xA</td>
</tr>
<tr>
<td></td>
<td>0xB</td>
</tr>
<tr>
<td></td>
<td>0xC</td>
</tr>
<tr>
<td>Application shared library data</td>
<td>0xD</td>
</tr>
<tr>
<td></td>
<td>0xE</td>
</tr>
<tr>
<td></td>
<td>0xF</td>
</tr>
</tbody>
</table>

- 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.
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 in the documentation for your release of AIX. For example: [Large program support]

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 the mmap() or shmat() methods, it is clear whether memory allocated to a specific segment of memory (under Esid) is allocated to the Java heap or the native heap. Use the type and description fields for each of the segments to determine which sections are native or Java heap. Segments that are allocated by using the mmap() or shmat() methods are listed as mmap mapped to or extended shm segments, and are in the Java heap. Segments that are allocated by using malloc are marked as working storage, and are in the native heap. You can use this demarcation 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:
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; other JVM loc P: 0x2BA0028-0x2BA0113
  (2) Compiling java/lang/String.hashCode()I
  + (warm) java/lang/String.hashCode()I
  (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;
  (2) Compiling java/lang/String.charAt(I)C
  + (warm) java/lang/String.charAt(I)C
  (2) Compiling java/util/Locale.toLowerCase(Ljava/lang/String;)Ljava/lang/String;
  + (warm) java/util/Locale.toLowerCase(Ljava/lang/String;)Ljava/lang/String;

Chapter 10. Troubleshooting and support  277
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 the documentation for your version of AIX. For example, for version 7.1: [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 diagnostic data” on page 436 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 Java heap is exhausted, an error message is received indicating an OutOfMemoryError condition with the Java heap.

If the process address space (that is, the native heap) is exhausted, an error message is received that explains that a native allocation has failed. In either case, the problem might not be a memory leak, just that the steady state of memory use that is required is higher than that available. Therefore, the first step is to determine which heap is being exhausted and increase the size of that heap.
If the problem is occurring because of a real memory leak, increasing the heap size does not solve the problem, but does delay the onset of the OutOfMemoryError exception or error conditions. That delay can be helpful on production systems.

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

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

**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 ["JVM messages" on page 595](#) 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 69](#) 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 69](#) 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 363](#). 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 3.1Malloc 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

Tracing leaks:

Some useful techniques for tracing leaks are built into the JVM.

The techniques are:

- The -verbose:gc option.
- HPROF tools. See “Using the HPROF Profiler” on page 491.

Tracing application use of direct byte buffers:

You can use the trace facility to diagnose the cause of excessive memory usage or OutOfMemoryError exceptions from applications that use direct byte buffers.

Trace points are available to help diagnose memory problems associated with the use of direct byte buffers. The trace point IDs are j9jcl.335 to j9jcl.338, and have the following form:

- Trace point j9jcl.335 prints the amount of memory being allocated:
  >sun_misc_Unsafe_allocateDBBMemory(0%xzx)
- Trace point j9jcl.336 prints when memory cannot be allocated:
  <sun_misc_Unsafe_allocateDBBMemory -- OutOfMemory
- Trace point j9jcl.337 prints the address of the allocated memory:
  <sun_misc_Unsafe_allocateDBBMemory result = %p
- Trace point j9jcl.338 prints the address of the memory being freed:
  >sun_misc_Unsafe_freeDBBMemory(%p)

Note: Trace point IDs are subject to change without notice. To achieve reliable results, see “Determining the tracepoint ID of a tracepoint” on page 419.

The trace point IDs can be used with the -Xtrace option to track down problems within a component. The -Xtrace command can direct output to a file or the console, or to internal buffers, which are dumped to a file when a problem occurs. There are many options associated with the trace facility that can be used to diagnose problems. See the section “Tracing Java applications and the JVM” on page 393. For specific information about setting -Xtrace options, see “Controlling the trace” on page 398.

For example, to generate console output when the trace points are called, use the command:
-Xtrace:print=j9jcl.335-338

The output generated is similar to:

```
17:41:05.420 0x61fa00 j9jcl.335 > sun_misc_Unsafe_allocateDBBMemory(0x21d8)
17:41:05.421 0x61fa00 j9jcl.337 < sun_misc_Unsafe_allocateDBBMemory result = 6871CC10
17:41:05.428+0x6b926600 j9jcl.338 > sun_misc_Unsafe_freeDBBMemory(6871CC10)
```

You can also include stack traces in the console output with the command:
-Xtrace:print=j9jcl.335-338,trigger=tpnid{j9jcl.335-338},jstacktrace

Here is an example that includes stack trace output, generated by the command:
-Xtrace:print=j9jcl.335-338,trigger=tpnid{j9jcl.335,jstacktrace},trigger=tpnid{j9jcl.338,jstacktrace}
You can use the -Xrunjnichk option to trace JNI calls that are made by your JNI code or by any JVM components that use JNI. This helps you to identify incorrect uses of JNI libraries from native code and can help you to diagnose JNI memory leaks.

JNI memory leaks occur when a JNI thread allocates objects and fails to free them. The Garbage Collector does not have enough information about the JNI thread to know when the object is no longer needed. For more information, see “The JNI and the Garbage Collector” on page 133.

Note that -Xrunjnichk is equivalent to -Xcheck:jni. See “Debugging the JNI” on page 140 for information on the -Xrunjnichk suboptions.

The -Xcheck:memory option can help you identify memory leaks inside the JVM. The -Xcheck:memory option traces the JVM calls to the operating system's malloc() and free() functions, and identifies any JVM mistakes in memory allocation.

The system property -Dcom.ibm.dbgmalloc=true provides memory allocation information about class library native code. Use this system property with the -Xcheck:memory:callsite=1000 option to obtain detailed information about class library callsites and their allocation sizes. Here is some sample output:

```
For more information about setting -Dcom.ibm.dbgmalloc=true, see “System property command-line options” on page 527.
```
For more information about the \texttt{-Xcheck:memory} option, see "JVM command-line options" on page 549.

Using Heapdump to debug memory leaks:

You can use Heapdump to analyze the Java Heap.

For details about analyzing the Heap, see "Using Heapdump" on page 363.

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 253, 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. \texttt{-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:

<table>
<thead>
<tr>
<th>kthr</th>
<th>memory</th>
<th>page</th>
<th>faults</th>
<th>cpu</th>
</tr>
</thead>
<tbody>
<tr>
<td>r b a</td>
<td>fre re pi po fr sr cy in sy cs us sy id wa</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 189898 612 0 0 0 3 11 0 178 606 424 6 1 92 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 189898 611 0 1 0 0 0 0 114 4573 122 96 4 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 189898 611 0 0 0 0 0 0 115 420 102 99 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 189898 611 0 0 0 0 0 0 114 425 91 99 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 189898 611 0 0 0 0 0 0 114 428 90 99 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 189898 610 0 1 0 0 0 0 117 333 102 97 3 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 189898 610 0 0 0 0 0 0 114 433 91 99 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 189898 610 0 0 0 0 0 0 114 429 94 99 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 189898 610 0 0 0 0 0 0 115 437 94 99 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 189898 609 0 1 0 0 0 0 116 340 99 98 2 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The previous example 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 258.

**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

<table>
<thead>
<tr>
<th>Process</th>
<th>Freq</th>
<th>Total Kernel</th>
<th>User</th>
<th>Shared</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>****</td>
<td>****</td>
<td>****</td>
<td>****</td>
<td>****</td>
</tr>
<tr>
<td>./java</td>
<td>5</td>
<td>59.39</td>
<td>24.28</td>
<td>0.00</td>
<td>35.11</td>
</tr>
<tr>
<td>wait</td>
<td>4</td>
<td>40.33</td>
<td>40.33</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>/usr/bin/tprof</td>
<td>1</td>
<td>0.20</td>
<td>0.02</td>
<td>0.00</td>
<td>0.18</td>
</tr>
<tr>
<td>/etc/syncd</td>
<td>3</td>
<td>0.05</td>
<td>0.05</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>/usr/bin/sh</td>
<td>2</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>gil</td>
<td>2</td>
<td>0.01</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>afsd</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>rpc.lockd</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>swapper</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>****</td>
<td>****</td>
<td>****</td>
<td>****</td>
<td>****</td>
</tr>
<tr>
<td>Total</td>
<td>20</td>
<td>100.00</td>
<td>64.70</td>
<td>0.00</td>
<td>35.29</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process</th>
<th>PID</th>
<th>TID</th>
<th>Total Kernel</th>
<th>User</th>
<th>Shared</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>****</td>
<td>****</td>
<td>****</td>
<td>****</td>
<td>****</td>
<td>****</td>
</tr>
<tr>
<td>./java</td>
<td>467018</td>
<td>819317</td>
<td>16.68</td>
<td>5.55</td>
<td>0.00</td>
<td>11.13</td>
</tr>
<tr>
<td>./java</td>
<td>467018</td>
<td>766019</td>
<td>14.30</td>
<td>6.30</td>
<td>0.00</td>
<td>8.00</td>
</tr>
<tr>
<td>./java</td>
<td>467018</td>
<td>725211</td>
<td>14.28</td>
<td>6.24</td>
<td>0.00</td>
<td>8.04</td>
</tr>
<tr>
<td>./java</td>
<td>467018</td>
<td>712827</td>
<td>14.11</td>
<td>6.16</td>
<td>0.00</td>
<td>7.94</td>
</tr>
</tbody>
</table>
```
```plaintext
| 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/tpref | 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 | 61471 | 0.00  | 0.00  | 0.00 | 0.00 | 0.00 |
| /usr/bin/sh | 397320 | 839883 | 0.03  | 0.02  | 0.00 | 0.00 | 0.00 |
| gil | 53274 | 73765 | 0.00  | 0.00  | 0.00 | 0.00 | 0.00 |
| /usr/bin/sh | 397318 | 839881 | 0.03  | 0.02  | 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 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 `tpref` output can be analyzed.

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

Chapter 10. Troubleshooting and support 285
This section shows that almost all of the time is being spent in one particular shared library, which is part of the JVM installation: `libj9gc<vm_version>.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:

- **libbcv<vm_version>.so**
  *Bytecode Verifier*

- **libdbg<vm_version>.so**
  *Debug Server (used by the Java Debug Interface)*

- **libj9gc<vm_version>.so**
  *Garbage Collection*

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

- **libj9jit<vm_version>.so**
  *The Just In Time (JIT) Compiler*

- **libj9jvmti<vm_version>.so**
  *The JVMTI interface*

- **libj9prt<vm_version>.so**
  *The "port layer" between the JVM and the Operating System*

- **libj9shr<vm_version>.so**
  *The shared classes library*

- **libj9thr<vm_version>.so**
  *The threading library*

- **libj9ute<vm_version>.so**
  *The trace engine*

- **libj9vm<vm_version>.so**
  *The core Virtual Machine*

- **libj9zlib<vm_version>.so**
  *The compressed file utility library*

- **libjclscar<vm_version>.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 `libj9gc<vm_version>.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

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>%</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheme::scanMixedObject(MM_Environment*,J9Object*)</td>
<td>2.67</td>
<td>MarkingScheme.cpp</td>
</tr>
<tr>
<td>MarkingScheme::scanClass(MM_Environment*,J9Class*)</td>
<td>2.54</td>
<td>MarkingScheme.cpp</td>
</tr>
<tr>
<td>.GC_ConstantPoolObjectSlotIterator::nextSlot()</td>
<td>1.96</td>
<td>ObjectSlotIterator.cpp</td>
</tr>
<tr>
<td>MarkingScheme::scanMixedObject(MM_Environment*,J9Object*)</td>
<td>1.96</td>
<td>MarkingScheme.cpp</td>
</tr>
<tr>
<td>lelTask::handleNextWorkUnit(MM_EnvironmentModron*)</td>
<td>1.05</td>
<td>ParallelTask.cpp</td>
</tr>
<tr>
<td>socket::getPacket(MM_Environment*,MM_Packet**)</td>
<td>0.70</td>
<td>WorkPackets.cpp</td>
</tr>
<tr>
<td>scheme::fixupRegion(J9Object*,J9Object*,bool,long&amp;)</td>
<td>0.67</td>
<td>CompactScheme.cpp</td>
</tr>
<tr>
<td>WorkPackets::putPacket(MM_Environment*,MM_Packet*)</td>
<td>0.47</td>
<td>WorkPackets.cpp</td>
</tr>
<tr>
<td>rkingScheme::scanObject(MM_Environment*,J9Object*)</td>
<td>0.43</td>
<td>MarkingScheme.cpp</td>
</tr>
<tr>
<td>sweepChunk(MM_Environment*,MM_ParallelSweepChunk*)</td>
<td>0.42</td>
<td>ParallelSweepScheme.cpp</td>
</tr>
<tr>
<td>MarkingScheme::scanMixedObject(MM_Environment*,J9Object*)</td>
<td>0.38</td>
<td>MarkingScheme.cpp</td>
</tr>
<tr>
<td>M_CompactScheme::getForwardingPtr(J9Object*) const</td>
<td>0.36</td>
<td>CompactScheme.cpp</td>
</tr>
<tr>
<td>ObjectHeapIteratorAddressOrderedList::nextObject()</td>
<td>0.33</td>
<td>dressOrderedList.cpp</td>
</tr>
<tr>
<td>.MM_HeapVirtualMemory::getHeapBase()</td>
<td>0.23</td>
<td>eapVirtualMemory.cpp</td>
</tr>
</tbody>
</table>

This output shows that the most-used functions are:

MarkingScheme::scanMixedObject(MM_Environment*,J9Object*)
MarkingScheme::scanClass(MM_Environment*,J9Class*)
ObjectSlotIterator.GC_ConstantPoolObjectSlotIterator::nextSlot()
MarkingScheme::markClass(MM_Environment*,J9Class*)

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:

<table>
<thead>
<tr>
<th>Process</th>
<th>PID</th>
<th>TID</th>
<th>Total</th>
<th>Kernel</th>
<th>User</th>
<th>Shared</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>./java</td>
<td>7018</td>
<td>819317</td>
<td>16.68</td>
<td>5.55</td>
<td>0.00</td>
<td>11.13</td>
<td>0.00</td>
</tr>
</tbody>
</table>
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:

```
3XMTTHREADINFO "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 `libj9gc<vm_version>.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  5054  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).
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 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 105 and “JIT and AOT problem determination” on page 430.

IBM Monitoring and Diagnostic Tools:

The IBM Monitoring and Diagnostic Tools are a set of GUI-based tools for monitoring applications and analyzing diagnostic data. These tools are designed to make diagnostic tasks as quick and as easy as possible.

Some tools can be attached to a running JVM, to monitor application behavior and resource usage. For other tools, you generate dump files from your system or JVM, then analyze the file in the tool. By using the tools, you can diagnose problems such as memory leaks, thread contention issues, and I/O bottlenecks, as well as getting information and recommendations to help you tune the JVM and improve the performance of your application.

For more information about the tools, see “Using the IBM Monitoring and Diagnostic Tools” on page 319.

Testing JVM optimizations:

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

Java monitor optimizations

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

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

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

Lock optimizations

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

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

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

**MustGather information for AIX**

The information that is most useful at a point of failure depends, in general, on the type of failure that is experienced.

The IBM Support Assistant Data Collector is the recommended utility for collecting Java diagnostics files for a problem event. The IBM Support Assistant Data Collector collects diagnostic files such as dumps and log files, and helps you to send the information to IBM, if required. For more information, see "IBM Support Assistant Data Collector" on page 436.

**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 255), 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 382. 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`.

**Java core file:**

When a java core file is written, a message (JVMDUMP010I) is written to STDERR telling you the name and full path of the java core file. In addition,
a java core 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-\.

JIT dump:
A general protection fault (GPF) or abort event generates a small binary dump of JIT diagnostic data. For more information, see “JIT dumps” on page 328.

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. The report might also point to the source of the problem where it is otherwise ambiguous.

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 292
- “Common NLS problem and possible causes” on page 293

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 items:
For Swing components:
1. Check your locale with locale
2. To change the locale, export LANG=zh_TW (for example)
3. If you know which font you have used in your application, such as serif,
   try to get the corresponding physical font by looking in the fontpath. If the
   font file is missing, try adding it there.

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 xlfld <physical font name> command to check whether the
system can recognize the character or not.

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” on page 294
• “Debug properties” on page 295
• “ORB exceptions” on page 296
• “Completion status and minor codes” on page 297
• “Java security permissions for the ORB” on page 298
• “Interpreting the stack trace” on page 299
Identifying an ORB problem
A background of the constituents of the IBM ORB component.

What the ORB component contains
The ORB component contains the following items:

- Java ORB from IBM and rmi-iiop runtime environment (com.ibm.rmi.*, com.ibm.CORBA.*)
- RMI-IIOAP (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.rmiic.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 a performance degradation. First Failure Data Capture (FFDC) still works when tracing is turned off, which means that serious errors are reported. If a debug file is produced, examine it for issues. For example, the server might have stopped without performing an ORB.shutdown().

You can use the following properties to enable the ORB traces:

• com.ibm.CORBA.Debug:

<table>
<thead>
<tr>
<th>Property value</th>
<th>Trace output information</th>
</tr>
</thead>
<tbody>
<tr>
<td>false [default]</td>
<td>Tracing is disabled, so no information is recorded.</td>
</tr>
<tr>
<td>true</td>
<td>Tracing is enabled. The output contains messages and traces for the entire ORB code flow</td>
</tr>
</tbody>
</table>

Note: If you use this property without specifying a value, tracing is enabled.

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

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

Here is an example of common usage:
java -Dcom.ibm.CORBA.Debug=true -Dcom.ibm.CORBA.Debug.Component=DISPATCH
-Dcom.ibm.CORBA.Debug.Output=trace.log
-Dcom.ibm.CORBA.CommTrace=true <classname>

Note: The example provided is a single line, entered on the command line.

• 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.Debug.Component=DISPATCH
  -Dcom.ibm.CORBA.CommTrace=true <classname>
  ```

**Note:** The example provided is a single line, entered on the command line.

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
```

**Related reference:**

- **“-Dcom.ibm.CORBA.Debug” on page 529**
  This system property enables debugging for the Object Request Broker (ORB) and includes tracing options that control how much information is recorded.

- **“-Dcom.ibm.CORBA.Debug.Component” on page 529**
  This system property can be used with `-Dcom.ibm.CORBA.Debug=true` to generate trace output for specific Object Request Broker (ORB) subcomponents such as MARSHAL or DISPATCH. This finer level of tracing helps you debug problems with ORB operations.

- **“-Dcom.ibm.CORBA.Debug.Output” on page 530**
  This system property redirects Object Request Broker (ORB) trace output to a file, which is known as a trace log.

- **“-Dcom.ibm.CORBA.CommTrace” on page 528**
  This system property turns on wire tracing for the Object Request Broker (ORB), which is also known as Comm tracing.

**ORB exceptions**

The exceptions that can be thrown are split into user and system categories.

If your problem is related to the ORB, unless your application is doing nothing or giving you the wrong result, your log file or terminal is probably full of exceptions that include the words “CORBA” and “rmi” many times. All unusual behavior that occurs in a good application is highlighted by an exception. This principle also applies for the ORB with its CORBA exceptions. Similarly to Java, CORBA divides its exceptions into user exceptions and system exceptions.
User exceptions

User exceptions are IDL defined and inherit from org.omg.CORBA.UserException. These exceptions are mapped to checked exceptions in Java; that is, if a remote method raises one of them, the application that called that method must catch the exception. User exceptions are usually not unrecoverable exceptions and should always be handled by the application. Therefore, if you get one of these user exceptions, you know where the problem is, because the application developer had to make allowance for such an exception to occur. In most of these cases, the ORB is not the source of the problem.

System exceptions

System exceptions are thrown transparently to the application and represent an unusual condition in which the ORB cannot recover gracefully, such as when a connection is dropped. The CORBA 2.6 specification defines 31 system exceptions and their mapping to Java. They all belong to the org.omg.CORBA package. The CORBA specification defines the meaning of these exceptions and describes the conditions in which they are thrown.

The most common system exceptions are:

- **BAD_OPERATION**: This exception is thrown when an object reference denotes an existing object, but the object does not support the operation that was called.
- **BAD_PARAM**: This exception is thrown when a parameter that is passed to a call is out of range or otherwise considered not valid. An ORB might raise this exception if null values or null pointers are passed to an operation.
- **COMM_FAILURE**: This exception is raised if communication is lost while an operation is in progress, after the request was sent by the client, but before the reply from the server has been returned to the client.
- **DATA_CONVERSION**: This exception is raised if an ORB cannot convert the marshaled representation of data into its native representation, or cannot convert the native representation of data into its marshaled representation. For example, this exception can be raised if wide character codeset conversion fails, or if an ORB cannot convert floating point values between different representations.
- **MARSHAL**: This exception indicates that the request or reply from the network is structurally not valid. This error typically indicates a bug in either the client-side or server-side run time. 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 596 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:

<table>
<thead>
<tr>
<th>Class/Interface</th>
<th>Method</th>
<th>Required permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.omg.CORBA.ORB</td>
<td>init</td>
<td>java.net.SocketPermission resolve</td>
</tr>
<tr>
<td>org.omg.CORBA.ORB</td>
<td>connect</td>
<td>java.net.SocketPermission listen</td>
</tr>
<tr>
<td>org.omg.CORBA.ORB</td>
<td>resolve_initial_references</td>
<td>java.net.SocketPermission connect</td>
</tr>
<tr>
<td>org.omg.CORBA.portable.ObjectImpl</td>
<td>_is_a</td>
<td>java.net.SocketPermission connect</td>
</tr>
</tbody>
</table>
### Class/Interface

<table>
<thead>
<tr>
<th>Class/Interface</th>
<th>Method</th>
<th>Required permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.omg.CORBA.portable.ObjectImpl</td>
<td>_non_existent</td>
<td>java.net.SocketPermission</td>
</tr>
<tr>
<td></td>
<td></td>
<td>connect</td>
</tr>
<tr>
<td>org.omg.CORBA.portable.ObjectImpl</td>
<td>OutputStream _request (String, boolean)</td>
<td>java.net.SocketPermission</td>
</tr>
<tr>
<td></td>
<td></td>
<td>connect</td>
</tr>
<tr>
<td>org.omg.CORBA.portable.ObjectImpl</td>
<td>_get_interface_def</td>
<td>java.net.SocketPermission</td>
</tr>
<tr>
<td></td>
<td></td>
<td>connect</td>
</tr>
<tr>
<td>org.omg.CORBA.Request</td>
<td>invoke</td>
<td>java.net.SocketPermission</td>
</tr>
<tr>
<td></td>
<td></td>
<td>connect</td>
</tr>
<tr>
<td>org.omg.CORBA.Request</td>
<td>send_defered</td>
<td>java.net.SocketPermission</td>
</tr>
<tr>
<td></td>
<td></td>
<td>connect</td>
</tr>
<tr>
<td>org.omg.CORBA.Request</td>
<td>send_oneway</td>
<td>java.net.SocketPermission</td>
</tr>
<tr>
<td></td>
<td></td>
<td>connect</td>
</tr>
<tr>
<td>javax.rmi.PortableRemoteObject</td>
<td>narrow</td>
<td>java.net.SocketPermission</td>
</tr>
<tr>
<td></td>
<td></td>
<td>connect</td>
</tr>
</tbody>
</table>

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:

```
completed: No
at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:199)
at com.ibm.rmi.iop.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.iop.CDRInputStream.read_value(CDRInputStream.java:1429)
at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie._invoke(_EJSRemoteStatelessPmiService_Tie.java:613)
at com.ibm.CORBA.iop.ExtensionsDelegate.dispatch(ExtensionsDelegate.java:515)
at com.ibm.CORBA.iop.ORB.process(ORB.java:2377)
at com.ibm.CORBA.iop.ORBWorker.run(ORBWorker.java:186)
at com.ibm.ejs.oa.pool.ThreadPool$PooledWorker.run(ThreadPool.java:104)
at com.ibm.ws.util.CachedThread.run(ThreadPool.java:137)
```

In the example, the ORB mapped a Java exception to a CORBA exception. This exception is sent back to the client later as part of a reply message. The client ORB reads this exception from the reply. It maps it to a Java exception (java.rmi.RemoteException according to the CORBA specification) and throws this new exception back to the client application.
Along this chain of events, often the original exception becomes hidden or lost, as does its stack trace. On early versions of the ORB (for example, 1.2.x, 1.3.0) the only way to get the original exception stack trace was to set some ORB debugging properties. Newer versions have built-in mechanisms by which all the nested stack traces are either recorded or copied around in a message string. When dealing with an old ORB release (1.3.0 and earlier), it is a good idea to test the problem on newer versions. Either the problem is not reproducible (known bug already solved) or the debugging information that you obtain is much more useful.

**Description string:**

The example stack trace shows that the application has caught a CORBA org.omg.CORBA.MARSHAL system exception. After the MARSHAL exception, some extra information is provided in the form of a string. This string should specify minor code, completion status, and other information that is related to the problem. Because CORBA system exceptions are alarm bells for an unusual condition, they also hide inside what the real exception was.

Usually, the type of the exception is written in the message string of the CORBA exception. The trace shows that the application was reading a value (read_value()) when an IllegalAccessException occurred that was associated to class com.ibm.ws.pmi.server.DataDescriptor. This information is an indication of the real problem and should be investigated first.

**Interpreting ORB traces**

The ORB trace file contains messages, trace points, and wire tracing. This section describes the various types of trace.

**ORB trace message:**

An example of an ORB trace message.

Here is a simple example of a message:

```
```

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;0=0;port=1360 ORBRas[default] Entry
.....
14:54:14.857 com.ibm.rmi.iiop.Connection <init>:539 LT=0;P=650241;0=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:

```plaintext
// 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 (0x10B)
00000000 00000028 4944443A 6F6D672E
6F72672F 53656E64 696E6743 6F6E7465
78742F43 6F646542 6173653A 312E3000
00000001 00000000 0000006C 00010200
00000000 392E3230 2E313738 2E313336
00001324 0000001A 4C4D4249 00000010
15074A96 00100000 00000000 00000000
```
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 00000001A 4C4D4249  ..........LMBI
0020: 0020: 0000010 14F94CA4 00100000 00000000   ...........L
0030: 0030: 00000000 00000000 00000008 6D657373  ..........mess
0040: 0040: 61676500 00000003 49424D12 00000008   age.....IBM....
0050: 0050: 0050: 00000000 14000005 00000001   14000005
0060: 0060: 0060: 00000000 00100100 00000006   00100100
0070: 0070: 0070: 000000A8 00000000 00000801   ...................
0080: 0080: 0080: 00000000 14000005 00000000   14000005
0090: 0090: 0090: 00000000 00010200 0000000D   00010200
00A0: 00A0: 00A0: 00000001 00000007 392E3230  1.0............l
00B0: 00B0: 00B0: 00010200 00000000 392E3230  9.20.178
00C0: 00C0: 00C0: 00010200 00000000 392E3230  9.20.178
00D0: 00D0: 00D0: 00010200 00000000 392E3230  9.20.178
00E0: 00E0: 00E0: 00010200 00000000 392E3230  9.20.178
00F0: 00F0: 00F0: 00010200 00000000 392E3230  9.20.178
0100: 00100100 00000000 00000000 00000000   ..........IBM.
0110: 00000008 00000000 14000005 00000000   14000005

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: https://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 112.](#)

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 IBM Z 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 341.

In general, stop the application, enable the orb traces and restart the application. When the hang is reproduced, the partial traces that can be retrieved can be used by the IBM ORB service team to help understand where the problem is.

**Starting the client before the server is running:**

If the client is started before the server is running, an error occurs when you run the client.

An example of the error messages that are generated from this process.

This operation outputs:

```
(org.omg.CORBA.COMM_FAILURE)
Hello Client
exception:
org.omg.CORBA.COMM_FAILURE:minor code:1 completed:No
 at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:145)
 at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:77)
 at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
 at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:75)
 at com.ibm.rmi.corba.ClientDelegate.createQuery(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)
 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.java:197)
at com.ibm.rmi.corba.InitialNamingClient.resolve_initial_references(InitialNamingClient.java:252)
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 from the network.

Your output is:

(org.omg.CORBA.TRANSIENT CONNECT_FAILURE)
IBM ORB service: collecting data

This section describes how to collect data about ORB problems.

If after all these verifications, the problem is still present, collect at all nodes of the problem the following information:

- Operating system name and version.
- Output of `java -version`.
- Output of `rmic -iiop -version`, if rmic is involved.
- ASV build number (WebSphere Application Server only).
- If you think that the problem is a regression, include the version information for the most recent known working build and for the failing build.
- If this is a runtime problem, collect debug and communication traces of the failure from each node in the system (as explained earlier in this section).
- If the problem is in `rmic -iiop` or `rmic -idl`, set the options: `-J-Djavac.dump.stack=1 -Xtrace`, and capture the output.
- Typically this step is not necessary. If it looks like the problem is in the buffer fragmentation code, IBM service will return the defect asking for an additional set of traces, which you can produce by executing with `-Dcom.ibm.CORBA.FragmentSize=0`.

A testcase is not essential, initially. However, a working testcase that demonstrates the problem by using only the Java SDK classes will speed up the resolution time for the problem.

Preliminary tests:

The ORB is affected by problems with the underlying network, hardware, and JVM.

When a problem occurs, the ORB can throw an `org.omg.CORBA.*` exception, some text that describes the reason, a minor code, and a completion status. Before you assume that the ORB is the cause of problem, do the following checks:

- Check that the scenario can be reproduced in a similar configuration.
- Check that the JIT is disabled (see "JIT and AOT problem determination" on page 430).

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.

**Hardware compression acceleration issues on Power Systems**

Check that you have the required software and hardware, check that your device driver is working correctly, then use the trace function to check that your system is using hardware compression services and not software compression services. Analyze the trace output to find the point of failure.

You must have specific hardware and software components to use hardware compression services. You can read about the prerequisites in "Enabling hardware compression acceleration on Power Systems" on page 246.

You can turn on tracing with the environment variable ZLIB_TRACE. Use one of the following values:
- 0x1: provides general information
- 0x2: provides hardware information
- 0x4: provides software information
- 0x8: provides both hardware and software information
- 0xf: provides general, hardware, and software information

For example, to turn on trace and obtain general, hardware, and software information, specify the following command at a shell prompt:

```
export ZLIB_TRACE=0xf
```

Here is an example of trace output when an application is run in default mode, which uses software compression and decompression:

```plaintext
## init: BUILD=Feb 6 2014 ZLIB_TRACE=f ZLIB_INFLATE_IMPL=0 ZLIB_DEFLATE_IMPL=0
## [11def7b0] inflateInit2_ : windowBits=-15 version=1.2.7/1.2.7 stream_size=88 impl=0
## [11def7b0] inflate: flush=1 next_in=5fe546b8 avail_in=210 next_out=5fe546d0 avail_out=327 total_in=0 total_out=0
## [11def7b0] inflate: flush=1 next_in=5fe546b8 avail_in=210 next_out=5fe546d0 avail_out=327 total_in=0 total_out=0
## [11def7b0] inflate: flush=1 next_in=5fe546b10 avail_in=210 next_out=5fe546d0 avail_out=327 total_in=0 total_out=0
## [11def7b0] inflate: flush=1 next_in=5fe54be2 avail_in=0 next_out=5fe54ea7 avail_out=0 total_in=210 total_out=0
## [11def7b0] inflateReset
...
```

The first line in the trace shows that the environment variables ZLIB_INFLATE_IMPL and ZLIB_DEFLATE_IMPL are set to 0, which indicates that software compression and decompression mode is enabled. Subsequent lines show that the zlib software APIs, such as inflate, are in use. The trace includes information on the number of input and output bytes that are available, which can be useful for debugging problems.

Here is an example of trace output when an application is run in hardware compression and decompression mode:

```plaintext
## init: BUILD=Feb 6 2014 ZLIB_TRACE=f ZLIB_INFLATE_IMPL=1 ZLIB_DEFLATE_IMPL=1
## [11e0f7b0] inflateInit2_ : windowBits=-15 version=1.2.7/1.2.7 stream_size=88 impl=1
hh [11e0f7b0] h_inflateInit2_ : card_no=-1 ibuf_total=0
## [11e0f7b0] inflate: flush=1 next_in=5fe546b8 avail_in=0 next_out=5fe55560 avail_out=327 total_in=0
hh [11e0f7b0] h_inflate: flush=1 next_in=5fe546b8 avail_in=0 next_out=5fe55560 avail_out=327
hh [11e0f7b0] h_inflate: flush=1 next_in=5fe546b10 avail_in=0 next_out=5fe55560 avail_out=327 rc=0
## [11e0f7b0] inflate: flush=1 next_in=5fe55310 avail_in=210 next_out=5fe55560 avail_out=327 total_in=0 total_out=0
hh [11e0f7b0] h_inflate: flush=1 next_in=5fe55310 avail_in=210 next_out=5fe55560 avail_out=327
hh [11e0f7b0] h_inflate: flush=1 next_in=5fe5532e avail_in=0 next_out=5fe556a7 avail_out=0 total_in=210 total_out=1
## [11e0f7b0] inflateReset
hh [11e0f7b0] h_inflateReset
...
```

Chapter 10. Troubleshooting and support 307
The first line in the trace output shows the environment variables `ZLIB_INFLATE_IMPL` and `ZLIB_DEFLATE_IMPL` are set to 1, which indicates that hardware compression and decompression mode is enabled. Subsequent lines show that the hardware accelerated data compression APIs, such as `h_inflate`, are in use. To assist with debugging, the trace includes information on the number of input and output bytes available.

**Attach API problem determination**

Common problems and explanations that can help you solve issues that involve the Attach API.

The 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 virtual machine (VM) go to `stderr` or `stdout` for the target VM. They are not reported in the messages output by the attaching VM.

The following items describe some common problems and suggest, where possible, how these problems can be avoided:

- “Deleting or changing permissions on directories and files in `/tmp”
- “The `VirtualMachine.attach(String id)` method reports `AttachNotSupportedException: No provider for virtual machine id” on page 309
- “The `VirtualMachine.attach()` method reports `AttachNotSupportedException” on page 310
- “A process running as root can see a target that uses `AttachProvider.listVirtualMachines()` but attempting to attach to the target results in an `AttachNotSupportedException` exception” on page 310
- “The `/tmp/.com_ibm_tools_attach` directory contains many directories with numeric file names” on page 310
- “The target Java process is not reported in the list of virtual machines returned by `AttachProvider.listVirtualMachines()`” on page 311
- “The `/tmp/.com_ibm_tools_attach` directory contains many directories with file names that contain underscores” on page 311
- “The shared semaphore is not removed when a Java virtual machine ends” on page 311

**Deleting or changing permissions on directories and files in `/tmp”**

The Attach API depends on the contents of a common directory. By default the common directory is `/tmp/.com_ibm_tools_attach`, but you can specify a different directory by using the `-Dcom.ibm.tools.attach.directory` system property. The common directory must have owner, group, and world read, write, and execute permissions, and the sticky bit must be set. The common files `_attachmentlock`, `_master`, and `_notifier` must have owner, group, and world read and write permissions. Execute permissions are not required.
Problems are caused if you modify the common directory in one of the following ways:

- Deleting the common directory.
- Deleting the contents of the common directory.
- Changing the permissions of the common directory or any of its content.

If you do modify the common directory, possible effects include:

- Semaphore “leaks” might occur, where excessive numbers of unused shared semaphores are opened. You can remove the semaphores by using the command:

  \texttt{ipcrm -s <semid>}

  Use the command to delete semaphores that have keys that start with 0xa1.
- The VMs might not be able to list existing target VMs.
- The VMs might not be able to attach to existing target VMs.
- The VM might not be able to enable its Attach API.
- Java technology processes might not end, or might take an excessive length of time to end.

If the common directory cannot be used, the VM attempts to re-create the common directory. However, the JVM cannot re-create the files that are related to VMs that are currently running.

If the /tmp directory, or the directory that is specified by the \texttt{-Dcom.ibm.tools.attach.directory} system property, is full or inaccessible (for example, because of permissions), the Attach API fails to initialize and no error message is produced.

**The VirtualMachine.attach(String id) method reports AttachNotSupportedException: No provider for virtual machine id**

The following reasons might exist for this message:

- The target VM might be owned by another user ID. The Attach API can connect only a VM to a target VM with the same user ID.
- The Attach API for the target VM might not have fully initialized. A short delay can occur from when the Java VM starts to when the Attach API is functional.
- The Attach API for the target VM might have failed. Verify that the directory \texttt{/tmp/.com_ibm_tools_attach/<id>} exists, and that the directory is readable and writable by the user ID.
- The target directory \texttt{/tmp/.com_ibm_tools_attach/<id>} might have been deleted.
- The attach API common directory might be located on a network file system. This directory must be located on a locally-mounted file system. For more information, see [“Support for the Java Attach API” on page 195](#).
- 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:

  \texttt{ipcs -s}

  If a shared semaphore exists, at least one key starting with “0xa1” appears in the output from the \texttt{ipcs} command.

**Note:** The number of available semaphores is limited on systems that use System V IPC, including Linux, z/OS, and AIX.
The VirtualMachine.attach() method reports AttachNotSupportedException

The following reasons might exist 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 imposes a wait time on the port that is used to attach to the target. The wait time might occur after heavy use of the Attach API, or for other protocols that use sockets. To check if any ports are in the TIME_WAIT state, use the command:

```bash
netstat -a
```

The VirtualMachine.loadAgent(), VirtualMachine.loadAgentLibrary(), or VirtualMachine.loadAgentPath() methods report com.sun.tools.attach.AgentLoadException or com.sun.tools.attach.AgentInitializationException

The following reasons might exist for this message:

- The JVMTI agent or the agent JAR file might be corrupted. Try loading the agent at startup time by using the -javaagent, -agentlib, or -agentpath option, depending on which method reported the problem.
- The agent might be attempting an operation that is not available after VM startup.

A process running as root can see a target that uses AttachProvider.listVirtualMachines(), but attempting to attach to the target results in an AttachNotSupportedException exception

A process can attach only to processes owned by the same user. To attach to a non-root process from a root process, first use the su command to change the effective UID of the attaching process to the UID of the target UID, before attempting to attach.

The /tmp/.com_ibm_tools_attach directory contains many directories with numeric file names

Each Java technology process creates a private directory in the /tmp/.com_ibm_tools_attach directory, by using its process ID as the directory name. When the process ends, it deletes this private directory. If the process, or the operating system, crashes, or you end the process by using the SIGKILL command, these obsolete directories are not removed.

Subsequent processes delete obsolete directories automatically over time. Each process examines a sample of the directories in /tmp/.com_ibm_tools_attach. If a directory is obsolete and is owned by the user that is running the process, the process deletes that directory.

To force deletion of all obsolete directories that are owned by the current user, run the jconsole command, found in the SDK bin directory. When the New Connection dialog is displayed, click Cancel, then exit the application.
The Attach API attempts to delete unused files and directories but cannot delete files that belong to another user. These files are deleted when that user, or ROOT user, runs a Java program.

To clean up all obsolete directories for all users, run the `jconsole` command as the root user.

**The target Java process is not reported in the list of virtual machines returned by AttachProvider.listVirtualMachines()**

The following reasons might exist for this problem:

- On the z/OS operating system, the Attach API is turned off by default and must be enabled explicitly by setting the `com.ibm.tools.attach.enable` system property to `yes`.
- The Attach API for the target process might be disabled. Check that the setting for the `com.ibm.tools.attach.enable` system property is set to `yes`.
- A file access issue might be preventing the Attach API initializing for the target process.

The following causes might exist for this problem:

- The process is unable to read and write to the common directory, which is typically named `.com.ibm.tools.attach`. This directory can be found in the directory denoted by the `java.io.tmpdir` system property.
- A process requires a target directory in the common directory with the same name as its virtual machine ID, which is typically the process ID. If this directory exists but is not accessible, the process is not available for late attach. Such a directory can exist if a Java process for another user failed or was sent a `SIGKILL` signal.
- The process was unable to create or access other resources such as lock files or shared semaphores.

**The `/tmp/.com.ibm.tools_attach` directory contains many directories with file names that contain underscores**

If a process is unable to create a target directory because a directory of the same name exists and cannot be deleted, the process tries to create a limited number of variations to the name.

For example, if the process 1234 cannot create the directory 1234, the process tries to create 1234_0, 1234_1, and so on.

**The shared semaphore is not removed when a Java virtual machine ends**

On shutdown, the Java process removes the shared semaphore only if it can determine that the semaphore is not required by other processes. Under race conditions, a shared semaphore might exist when no Java processes are running. In this case, the semaphore is reused by the next Java process. If the common directory is deleted while a Java process is running, an extra semaphore with a key that starts with 0xa1 might be created. These semaphores can be removed manually if no Java processes are running.

**Related concepts:**
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.

**Related reference:**

*“-Dcom.ibm.tools.attach.enable” on page 536*

Enable the Attach API for this application.

### Using diagnostic tools

Diagnostic tools are available to help you solve your problems.

This section describes how to use the tools. The chapters are:

- “Overview of the available diagnostic tools”
- “Using dump agents” on page 321
- “Using Javadump” on page 340
- “Using Heapdump” on page 363
- “Using system dumps and the dump viewer” on page 373
- “Tracing Java applications and the JVM” on page 393
- “JIT and AOT problem determination” on page 430
- “Garbage Collector diagnostic data” on page 436
- “Class-loader diagnostic data” on page 460
- “Shared classes diagnostic data” on page 463
- “Using the HPROF Profiler” on page 491
- “Using the JVMTI” on page 496
- “Using the Diagnostic Tool Framework for Java” on page 513
- “Using JConsole” on page 520

### Overview of the available diagnostic tools

The diagnostic information that can be produced by the JVM is described in the following topics. A range of supplied tools can be used to post-process this information and help with problem determination.

Subsequent topics in this part of the product documentation 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 virtual machine 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 321). Older versions of the IBM virtual machine 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 599 lists the supported environment variables.

The format of the various types of diagnostic information produced is specific to the IBM virtual machine and might change between releases of the JVM.

The types of diagnostic information that can be produced are:

Javadoc
The Javadoc is sometimes referred to as a Javacore or thread dump in some JVMs. This dump is in a human-readable format produced by default when the JVM terminates unexpectedly because of an operating system signal, an OutOfMemoryError exception, or when the user enters a reserved key combination (for example, Ctrl-Break on Windows). You can also generate a Javadoc by calling a method from the Dump API, for example com.ibm.jvm.Dump.JavaDump(), from inside the application. A Javadoc summarizes the state of the JVM at the instant the signal occurred. Much of the content of the Javadoc is specific to the IBM virtual machine. See “Using Javadoc” on page 340 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 ends 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 must be processed using available tools such as Memory Analyzer. See “Using Heapdump” on page 363 for more details.

System dumps
System dumps (also known as core dumps on Linux platforms) are platform-specific files that contain information about the active processes, threads, and system memory. System dumps are usually large. By default, system dumps are produced by the JVM only when the JVM fails.
unexpectedly because of a GPF (general protection fault) or a major JVM or system error. You can also request a system dump by using the Dump API. For example, you can call the com.ibm.jvm.Dump.SystemDump() method from your application. You can use the -Xdump:system option to produce system dumps when other events occur.

**JIT dumps**
When a general protection fault (GPF) or abort event occurs, the Just-In-Time (JIT) compiler produces a small binary dump of diagnostic data that can help IBM service troubleshoot problems. For more information about the location of this file, see “JIT dumps” on page 328.

**Garbage collection data**
A JVM started with the -verbose:gc option produces output in XML format that can be used to analyze problems in the Garbage Collector itself or problems in the design of user applications. Numerous other options affect the nature and amount of Garbage Collector diagnostic information produced. See “Garbage Collector diagnostic data” on page 436 for more information.

**Trace data**
The IBM virtual machine 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 393.

**Other data**
Special options are available for producing diagnostic information relating to
- The JIT (see “JIT and AOT problem determination” on page 430)
- Class loading (see “Class-loader diagnostic data” on page 460)
- Shared classes (see “Shared classes diagnostic data” on page 463)

You can also download the IBM Monitoring and Diagnostic Tools, a set of freely-available GUI-based tools for monitoring applications and analyzing diagnostic data. For more information, see “IBM Monitoring and Diagnostic Tools” on page 289.

**Summary of cross-platform tooling**
IBM has several cross-platform diagnostic tools. The following sections provide brief descriptions of the tools and indicate the different areas of problem determination to which they are suited.

**IBM Monitoring and Diagnostic Tools:**
The IBM Monitoring and Diagnostic Tools are a set of GUI-based tools for monitoring applications and analyzing diagnostic data. These tools are designed to make diagnostic tasks as quick and as easy as possible.

Some tools can be attached to a running JVM, to monitor application behavior and resource usage. For other tools, you generate dump files from your system or JVM,
then analyze the file in the tool. By using the tools, you can diagnose problems such as memory leaks, thread contention issues, and I/O bottlenecks, as well as getting information and recommendations to help you tune the JVM and improve the performance of your application.

For more information about the tools, see “Using the IBM Monitoring and Diagnostic Tools” on page 319.

**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 373.

**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 496. The HPROF tool provided with the SDK has been updated to use the JVMTI; see “Using the HPROF Profiler” on page 491.

**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 513.

Trace formatting:

JVM trace is a key diagnostic tool for the JVM. The IBM JVM incorporates a large degree of flexibility in determining what is traced and when it is traced. This flexibility enables you to tailor trace so that it has a relatively small effect on performance.

The IBM Virtual Machine for Java contains many embedded trace points. In this release, maximal tracing is enabled by default for a few level 1 tracepoints and exception trace points. Command-line options allow you to set exactly what is to be traced, and specify where the trace output is to go. Trace output is generally in an encoded format and requires a trace formatter to be viewed successfully.

In addition to the embedded trace points provided in the JVM code, you can place your own application trace points in your Java code. You can activate tracing for entry and exit against all methods in all classes. Alternatively, you can activate tracing for a selection of methods in a selection of classes. Application and method traces are interleaved in the trace buffers with the JVM embedded trace points. The tracing allows detailed analysis of the routes taken through the code.

Tracing is used mainly for performance and leak problem determination. Trace data might also provide clues to the state of a JVM before an unexpected termination or “hang”.

Trace and trace formatting are IBM-specific; that is, they are present only in the IBM Virtual Machine for Java. See “Using method trace” on page 423 and “Tracing Java applications and the JVM” on page 393 for more details. Although trace is not easy to understand, it is an effective tool.

Scenarios in which dumps might not be produced

In certain scenarios, a dump is not produced when a crash occurs. This section gives reasons why a dump is not produced and suggests how you can obtain a system dump.

A crash can occur with no dump produced. An example scenario is one in which the crash occurs during the shut down of the Java runtime environment. The Java runtime environment might not have time to produce all the debug information. In this case, the console output shows the start of the dump information, but the Java runtime environment cannot write the information in a dump file. For example, the console might show the following output:
Unhandled exception
Type=Segmentation error vmState=0x00000000
J9Generic_Signal_Number=00000004 ExceptionCode=c0000005 ExceptionAddress=430514B
E ContextFlags=0001003f
Handler1=7FEE9C40 Handler2=7FEC98C0 InaccessibleAddress=00000000
EDI=000A7060 ESI=43159598 EAX=00000000 EBX=001925EC
ECX=00000001 EDX=4368FECC
EIP=430514BE EBP=4368FED8 EFLAGS=00010246
Module=failing_module.dll
Module_base_address=43050000 Offset_in_DLL=000014BE
Target=2_40_20081203_026494_lHdSMr (Windows XP 5.1 build 2600 Service Pack 2)
CPU=x86 (2 Logical CPUs) (0x7fe6b000 RAM)

A diagnostic dump is not produced for several possible reasons. A common reason is that the Java runtime process was stopped by a user, a script, or by the operating system. Another possible reason is that the crash occurred on a JVM process that was very close to shut down, resulting in a race condition between the JVM dump handler and the main thread exiting the process.

Identifying if the race condition exists:
Enable trace points to check for situations in which no dump is produced after a crash.

About this task
Check for the situations in which no dump is produced after a crash by enabling trace points near the shut down of the Java runtime environment. If the trace points overlap with the crash condition, you have confirmation that the race condition occurred. The tracepoints in the protectedDestroyJavaVM are the last to be triggered before the main thread returns.

Procedure
1. Find the protectedDestroyJavaVM function tracepoints in the J9TraceFormat.dat file by using the instructions in “Determining the tracepoint ID of a tracepoint” on page 419.
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 - protectedDestroyJavaVM 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=000A7060 ESI=432235D8 EAX=00000000 EBX=00192684
ECX=00000001 EDX=4368FECC
EIP=430514BE 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)
protectedDestroyJavaVM GC HeapManagement ShutdownCPU=x86 (2 Logical CPUs) (0x7fe6b000 RAM)
```
The Unhandled exception message is printed after the first tracepoints for the protectedDestroyJavaVM function. This output shows that the crash occurred very late in the life of the Java runtime environment, and that enough time remained to produce the dumps before the process ended.

What to do next

When you confirm that a race condition has occurred, you might still be able to obtain a system dump. For more information, see “Obtaining system dumps in a race condition.”

Obtaining system dumps in a race condition:

You might be able to obtain system dumps even when a race condition exists.

About this task

When you confirm that you have a race condition in which shut down timing prevents a system dump, you can try to obtain a dump in two ways:
- Try to prevent the system from shutting down before the dump is taken.
- Add a delay near the end of the JVM run time to give the dump handler enough time to write the dumps.

Procedure

On AIX, z/OS, or Linux, create a system dump by using the -Xrs Java command-line option to disable the Java signal handler. The default signal handler in the operating system triggers a dump and prevents the system from shutting down before the dump is taken. For more information, see “Disabling dump agents with -Xrs and -Xrs:sync” on page 340.

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 255.

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:
In this case, follow the instructions in the “Disabling dump agents with -Xrs and -Xrs:sync” on page 340 section. The information helps you disable the JVM dump agents and configure the operating system to produce a system dump instead.

Related concepts:

“System resource limits and the ulimit command” on page 235

The operating system provides ways of limiting the amount of resource that can be used. Limits are set for each user, but are applied separately to each process that is running for that user. These limits can affect Java applications, for example if certain limits are too low, the system might not be able to generate a complete Java dump file.

Using the IBM Monitoring and Diagnostic Tools

The IBM Monitoring and Diagnostic Tools are a set of GUI-based tools which you can use to monitor your applications, analyze resource usage, and diagnose problems. The tools can help you to optimize application performance, improve application stability, reduce resource usage, and resolve problems more quickly.

The tools provide output in various formats, such as tables, charts, graphs, and recommendations. Use this output to complete the following diagnostic tasks:

- Detect deadlock conditions
- Monitor thread activity
- See which methods are taking the most time to run
- See which objects are using the most memory
- Find memory leaks and I/O bottlenecks
- Analyze the efficiency of Java collections, such as arrays
- Understand the relationships between application objects
- Visualize garbage collection performance
- Get recommendations for tuning the JVM and improving application performance

The following tools are available:

Health Center
Monitor a running JVM, with minimal performance overhead. Tuning recommendations are also provided.

Garbage Collection and Memory Visualizer
Analyze the memory usage, garbage collection behavior, and performance of applications, by plotting verbose garbage collection data from dump files. Tuning recommendations are also provided.

Interactive Diagnostic Data Explorer
Use commands to extract information from dump files. This tool is a GUI-based version of the jdmpview command, with extra features.

Memory Analyzer
Analyze the memory usage and performance of applications, using data from dump files.
The tools are available to download, free of charge, into IBM Support Assistant. Most tools are also available to download from Eclipse marketplace. The IBM Support Assistant is a free workbench that is designed to help you with problem determination. The IBM Monitoring and Diagnostic Tools is just one set of tools that you can install into the IBM Support Assistant.

For more information about the IBM Monitoring and Diagnostic Tools, see the IBM Monitoring and Diagnostic Tools product documentation on IBM Knowledge Center, and the IBM Monitoring and Diagnostic Tools developerWorks page.

**Garbage Collection and Memory Visualizer**

Garbage Collection and Memory Visualizer (GCMV) helps you understand memory use, garbage collection behavior, and performance of applications.

GCMV parses and plots data from various types of log, including the following types:

- Verbose garbage collection logs.
- Trace garbage collection logs, generated by using the -Xtgc parameter.
- Native memory logs, generated by using the ps, svmon, or perfmon system commands.

The tool helps to diagnose problems such as memory leaks, analyze data in various visual formats, and provides tuning recommendations.

GCMV is available from Eclipse marketplace and as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see: https://www.ibm.com/developerworks/java/jdk/tools/gcmv/.

Further information about GCMV is available in IBM Knowledge Center.

**Health Center**

Health Center is a diagnostic tool for monitoring the status of a running application.

The tool is provided in two parts:

- The Health Center agent that collects data from a running application.
- An Eclipse-based client that connects to the agent. The client interprets the data and provides recommendations to improve the performance of the monitored application.

Health Center is available from Eclipse marketplace and as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see: https://www.ibm.com/developerworks/java/jdk/tools/healthcenter/.

Further information about Health Center is available in IBM Knowledge Center.

**Interactive Diagnostic Data Explorer**

Interactive Diagnostic Data Explorer (IDDE) is a GUI-based alternative to the dump viewer (jdmpview command). IDDE provides the same functionality as the dump viewer, but with extra support such as the ability to save command output.

Use IDDE to more easily explore and examine dump files that are produced by your JVM. Within IDDE, you enter commands in an investigation log, to explore the dump file. The support that is provided by the investigation log includes the following items:
- Command assistance
- Auto-completion of text, and some parameters such as class names
- The ability to save commands and output, which you can then send to other people
- Highlighted text and flagging of issues
- The ability to add your own comments
- Support for using the Memory Analyzer from within IDDE

IDDE is available from Eclipse marketplace and as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see IDDE overview on developerWorks.

Further information about IDDE is available in IBM Knowledge Center.

Memory Analyzer
Memory Analyzer helps you analyze Java heaps using operating system level dumps and Portable Heap Dumps (PHD).

This tool can analyze dumps that contain millions of objects, providing the following information:
- The retained sizes of objects.
- Processes that are preventing the Garbage Collector from collecting objects.
- A report to automatically extract leak suspects.

This tool is based on the Eclipse Memory Analyzer (MAT) project, and uses the IBM Diagnostic Tool Framework for Java (DTFJ) feature to enable the processing of dumps from IBM JVMs.

Memory Analyzer is provided as an IBM Support Assistant (ISA) add-on. For information about installing and getting started with the add-on, see: https://www.ibm.com/developerworks/java/jdk/tools/memoryanalyzer/

Further information about Memory Analyzer is available in IBM Knowledge Center.

Using dump agents
Dump agents are set up during JVM initialization. They enable you to use events occurring in the JVM, such as Garbage Collection, thread start, or JVM termination, to initiate dumps or to start an external tool.

The default dump agents are sufficient for most cases. Use the -Xdump option to add and remove dump agents for various JVM events, update default dump settings (such as the dump name), and limit the number of dumps that are produced.

This section describes:
- "Using the -Xdump option" on page 322
- "Dump agents" on page 325
- "Dump events" on page 328
- "Advanced control of dump agents" on page 329
- "Dump agent tokens" on page 335
- "Default dump agents" on page 335
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 Java VM events.
- Update default dump agent settings.
- Limit the number of dumps produced.
- Show dump agent help.

The syntax of the -Xdump option is as follows:

-Xdump command-line option syntax

```
-Xdump: command-line option syntax
```

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

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

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

You can use the -Xdump:what option to list the registered dump agents. The registered dump agents listed might be different to the agents you specified. The difference is because the VM ensures that multiple -Xdump options are merged into a minimum set of dump agents.

The events keyword is used as the prime trigger mechanism. However, you can use additional keywords for further control of the dump produced.

The options that you can use with dump agents provide granular control. The following syntax applies:
-Xdump command-line agent option syntax

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

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

For more information, see the manual for your shell.

Help options

These options provide usage and configuration information for dumps, as shown in the following table:

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Xdump:help</td>
<td>Display general dump help</td>
</tr>
<tr>
<td>-Xdump:events</td>
<td>List available trigger events</td>
</tr>
<tr>
<td>-Xdump:request</td>
<td>List additional VM requests</td>
</tr>
<tr>
<td>-Xdump:tokens</td>
<td>List recognized label tokens</td>
</tr>
<tr>
<td>-Xdump:what</td>
<td>Show registered agents on startup</td>
</tr>
<tr>
<td>-Xdump:&lt;agent&gt;:help</td>
<td>Provides detailed dump agent help</td>
</tr>
<tr>
<td>-Xdump:&lt;agent&gt;:defaults</td>
<td>Provides default settings for this agent</td>
</tr>
</tbody>
</table>

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 \texttt{-Xdump:what} might not be grouped in the same way as the original \texttt{-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:

\begin{verbatim}
java -Xdump:java:events=unload -Xdump:what
\end{verbatim}

This command does not create a new agent, as can be seen in the results from the \texttt{-Xdump:what} option.

\begin{verbatim}
... \\
---------------------- \\
\texttt{-Xdump:java:} \\
\hspace{1em}events=gpf+user+abort+unload, \\
\hspace{1em}label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt, \\
\hspace{1em}range=1..0, \\
\hspace{1em}priority=10, \\
\hspace{1em}request=exclusive \\
---------------------- \\
\end{verbatim}

The configuration is merged with the existing javadump agent for events \texttt{gpf}, \texttt{user}, and \texttt{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:

\begin{verbatim}
java -Xdump:java:events=unload,priority=100 -Xdump:what
\end{verbatim}

The results of the \texttt{-Xdump:what} option in the command are as follows.

\begin{verbatim}
... \\
---------------------- \\
\texttt{-Xdump:java:} \\
\hspace{1em}events=unload, \\
\hspace{1em}label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt, \\
\hspace{1em}range=1..0, \\
\hspace{1em}priority=100, \\
\hspace{1em}request=exclusive \\
---------------------- \\
\texttt{-Xdump:java:} \\
\hspace{1em}events=gpf+user+abort, \\
\hspace{1em}label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt, \\
\hspace{1em}range=1..0, \\
\hspace{1em}priority=10, \\
\hspace{1em}request=exclusive \\
---------------------- \\
\end{verbatim}

To merge dump agents, the \texttt{request}, \texttt{filter}, \texttt{opts}, \texttt{label}, and \texttt{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:

\begin{verbatim}
java -Xdump:none -Xdump:java:events=uncaught,filter=java/lang/NullPointerException \ \
-Xdump:java:events=unload,filter=java/lang/NullPointerException -Xdump:what
\end{verbatim}

The results of this command are as follows.

Registered dump agents

\begin{verbatim}
---------------------- \\
\texttt{-Xdump:java:} \\
\hspace{1em}events=unload+uncaught, \\
\hspace{1em}filter=java/lang/NullPointerException, \\
\hspace{1em}label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt, \\
---------------------- \\
\end{verbatim}
Dump agents

A dump agent performs diagnostic tasks when triggered. Most dump agents save information on the state of the JVM for later analysis. The “tool” agent can be used to trigger interactive diagnostic data.

The following table shows the dump agents:

<table>
<thead>
<tr>
<th>Dump agent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td>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 326.</td>
</tr>
<tr>
<td>console</td>
<td>Basic thread dump to stderr.</td>
</tr>
<tr>
<td>system</td>
<td>Capture raw process image. See “Using system dumps and the dump viewer” on page 373.</td>
</tr>
<tr>
<td>tool</td>
<td>Run command-line program.</td>
</tr>
<tr>
<td>java</td>
<td>Write application summary. See “Using Javadump” on page 340.</td>
</tr>
<tr>
<td>heap</td>
<td>Capture heap graph. See “Using Heapdump” on page 363.</td>
</tr>
<tr>
<td>snap</td>
<td>Take a snap of the trace buffers.</td>
</tr>
<tr>
<td>jit</td>
<td>Capture JIT diagnostic data.</td>
</tr>
</tbody>
</table>

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
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 ^^^^^^^^^
JVMDUMP0013I 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
```

See “Using system dumps and the dump viewer” on page 373 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:

```
Thread=main (0188701C) Status=Running
at sun/misc/Resource.getBytes()Ljava/lang/byte[]{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$AppClassLoader.loadClass(Ljava/lang/Class;Ljava/lang/Class;Z)Ljava/lang/Class; (AppClassLoader.java:643)
at java/net/URLClassLoader$ClassFinder.run()Ljava/lang/Class; (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$AppClassLoader.loadClass(Ljava/lang/Class;Ljava/lang/Class;Z)Ljava/lang/Class; (AppClassLoader.java:643)
at sun/misc/Launcher$AppClassLoader.loadClass(Ljava/lang/Class;Ljava/lang/Class;Z)Ljava/lang/Class; (Launcher.java:345)
at java/lang/ClassLoader.loadClass(Ljava/lang/Class;)Ljava/lang/Class; (ClassLoader.java:609)
at TestLargeAllocations.main(LLjava/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 by specifying -Xdump:tokens. Alternatively, see the topic "Dump agent tokens” on page 335. 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 ...
```
By default, the range option is set to 1..1. If you do not specify a range option for the dump agent, the tool is started only once. To start the tool every time the event occurs, set the range option to 1..0. For more information, see “range option” on page 333.

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

**Javadumps:**

Javadumps are an internally generated and formatted analysis of the JVM, giving information that includes the Java threads present, the classes loaded, and heap statistics.

An example of producing a Javadump when a class is loaded:

```
java -Xdump:java:events=load,filter=java/lang/String -version
```

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

**Heapdumps:**

Heapdumps produce phd format files by default. “Using Heapdump” on page 363 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:none -Xdump:heap:events=vmstop,opts=PHD+CLASSIC
```

See “Using Heapdump” on page 363 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 “Running the trace formatter” on page 417 for more information about analyzing a snap trace.

JIT dumps:

The Just-In-Time (JIT) compiler produces a binary dump of diagnostic data when a general protection fault (GPF) or abort event occurs.

Messages similar to the following example indicate the location of the dump file:
```
JVMDUMP007I JVM Requesting JIT dump using '/home/user/jitdump.20130821.130125.3532.0004.dmp'
JVMDUMP010I JIT dump written to '/home/user/jitdump.20130821.130125.3532.0004.dmp'
```

Dump events

Dump agents are triggered by events occurring during JVM operation.

Some events can be filtered to improve the relevance of the output. For more information, see “filter option” on page 330.

Note: The gpf, traceassert, 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:

<table>
<thead>
<tr>
<th>Event</th>
<th>Triggered when...</th>
<th>Filter operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpf</td>
<td>A General Protection Fault (GPF) occurs.</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td>The JVM receives the SIGQUIT (Linux, AIX, z/OS, and i5/OS) or SIGBREAK (Windows) signal from the operating system.</td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td>The JVM receives the SIGABRT signal from the operating system.</td>
<td></td>
</tr>
<tr>
<td>vmstart</td>
<td>The virtual machine is started.</td>
<td></td>
</tr>
<tr>
<td>vmstop</td>
<td>The virtual machine stops.</td>
<td>Filters on exit code; for example, <code>filter=#129..#192#-42#255</code></td>
</tr>
<tr>
<td>load</td>
<td>A class is loaded.</td>
<td>Filters on class name; for example, <code>filter=java/lang/String</code></td>
</tr>
<tr>
<td>unload</td>
<td>A class is unloaded.</td>
<td></td>
</tr>
<tr>
<td>throw</td>
<td>An exception is thrown.</td>
<td>Filters on exception class name; for example, <code>filter=java/lang/OutOfMem*</code></td>
</tr>
<tr>
<td>Event</td>
<td>Triggered when...</td>
<td>Filter operation</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>catch</td>
<td>An exception is caught.</td>
<td>Filters on exception class name; for example, filter=<em>Memory</em></td>
</tr>
<tr>
<td>uncaught</td>
<td>A Java exception is not caught by the application.</td>
<td>Filters on exception class name; for example, filter=<em>MemoryError</em></td>
</tr>
<tr>
<td>systhrow</td>
<td>A Java exception is about to be thrown by the JVM.</td>
<td>Filters on exception class name; for example, filter=java/lang/OutOfMem*</td>
</tr>
<tr>
<td>thrstart</td>
<td>A new thread is started.</td>
<td></td>
</tr>
<tr>
<td>blocked</td>
<td>A thread becomes blocked.</td>
<td></td>
</tr>
<tr>
<td>thrstop</td>
<td>A thread stops.</td>
<td></td>
</tr>
<tr>
<td>fullgc</td>
<td>A garbage collection cycle is started.</td>
<td></td>
</tr>
<tr>
<td>slow</td>
<td>A thread takes longer than 50ms to respond to an internal JVM request.</td>
<td>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.</td>
</tr>
<tr>
<td>allocation</td>
<td>A Java object is allocated with a size matching the given filter specification</td>
<td>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.</td>
</tr>
<tr>
<td>traceassert</td>
<td>An internal error occurs in the JVM</td>
<td>Not applicable.</td>
</tr>
<tr>
<td>corruptcache</td>
<td>The JVM finds that the shared class cache is corrupt.</td>
<td>Not applicable.</td>
</tr>
<tr>
<td>excessivegc</td>
<td>An excessive amount of time is being spent in the garbage collector</td>
<td>Not applicable.</td>
</tr>
</tbody>
</table>

**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 326](#) 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 the specified dump type. For example:

```java
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 335](#) for more information.

The location for the dump is selected from these options, in this order:
1. The location specified by the `-Xdump:<agent>::file` option on the command line (if that location includes a path). This location applies to the specified dump agent type only.

2. The location specified by the `-Xdump:directory` option on the command line. This location applies to all dump agent types.

3. The location specified by the relevant environment variable.
   - `IBM_JAVACOREDIR` for Javadump.
   - `IBM_HEAPDUMPDIR` for Heapdump.
   - `IBM_COREDIR` for system dumps and JIT dumps.
   - `IBM_COREDIR` for snap traces.

4. 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 VM events occur thousands of times during the lifetime of an application. Dump agents can use filters and ranges to avoid excessive dumps being produced.

**Wildcards**

You can use a wildcard in your exception event filter by placing an asterisk only at the beginning or end of the filter. The following command does not work because the second asterisk is not at the end:

```
-Xdump:java:events=throw,filter=*
```

In order to make this filter work, it must be changed to:

```
-Xdump:java:events=throw,filter=*
```

**Class loading and exception events**

You can filter class loading (load) and exception (throw, catch, uncaught, systhrow) events by the name of the class that is being loaded, thrown or caught. For example:

```
-Xdump:java:events=load,filter=java/lang/String
-Xdump:java:events=throw,filter=java/lang/ArrayStoreException
-Xdump:java:events=catch,filter=java/lang/NullPointerException
```

In addition, you can filter throw, uncaught, and systhrow exception events by the name of the method that throws the exception. The name of the parent class must include the full package name, using the forward slash (/) as a separator. Use a dot (.) to separate the method name from the class name. You can use an asterisk (*) as a wildcard character, to include all methods. For example:

```
-Xdump:java:events=throw,filter=ExceptionClassName[com/ibm/ThrowingClassName.throwingMethodName[#stackFrameOffset]]
```

Optional portions are shown in brackets.
For example, to trigger a Javadump when method MyApplication.myMethod()
throws a NullPointerException, use the following syntax:

```
-Xdump:java:events=throw,filter=java/lang/NullPointerException#com/ibm/MyApplication.myMethod
```

The stack frame offset allows you to filter on the name of a method that calls the throwing method. This option is useful if the exception is being thrown from a general purpose or utility class. For example, to trigger a Javadump when a method called by MyApplication.main() throws a NullPointerException, use the following syntax:

```
-Xdump:java:events=throw,filter=java/lang/NullPointerException#com/ibm/MyApplication.main#1
```

The default value of the stack frame offset is zero.

You can filter the catch exception events by Java method name. For example:

```
-Xdump:java:events=catch,filter=ExceptionClassName[#com/ibm/CatchingClassName.catchingMethodName]
```

Optional portions are shown in brackets.

You can filter throw, uncaught, and syscall exception events by Java method name:

```
-Xdump:java:events=throw,filter=ExceptionClassName[#com/ibm/ThrowingClassName.throwingMethodName[#stackFrameOffset]]
```

Optional portions are shown in brackets.

You can filter the catch exception events by Java method name:

```
-Xdump:java:events=catch,filter=ExceptionClassName[#com/ibm/CatchingClassName.catchingMethodName]
```

Optional portions are shown in brackets.

**Note:** The filters apply to the stacktrace and fire every time the same exception is rethrown, which might result in multiple Java core files.

**vmstop event**

You can filter the JVM shut down event by using one or more exit codes:

```
-Xdump:java:events=vmstop,filter=#129..192#-42#255
```

**slow event**

You can filter the slow event to change the time threshold from the default of 50 ms:

```
-Xdump:java:events=slow,filter=#300ms
```

You cannot set the filter to a time that is less than the default time.

**Other events**

If you apply a filter to an event that does not support filtering, the filter is ignored.

**opts option:**

The Heapdump agent uses this option to specify the type of file to produce.
Heapdumps and the opts option

You can specify a PHD Heapdump, a classic text Heapdump, or both. For example:

- `Xdump:heap:opts=PHD` (default)
- `Xdump:heap:opts=CLASSIC`
- `Xdump:heap:opts=PHD+CLASSIC`

For more information, see “Enabling text formatted ("classic") Heapdumps” on page 364.

Tool dumps and the opts option

The tool dump agent supports two options that can be specified using the `opts` option. You can run the external process asynchronously with `opts=ASYNC`. You can also specify a delay in milliseconds that produces a pause after starting the command. These two options can be used independently or together. The following examples show different options for starting a new process that runs myProgram:

- `Xdump:tool:events=vmstop,exec=myProgram`

Without the `opts` option, the tool dump agent starts the process, and waits for the process to end before continuing.

- `Xdump:tool:events=vmstop,exec=myProgram,opts=ASYNC`

When `opts=ASYNC` is specified, the tool dump agent starts the process, and continues without waiting for the new process to end.

- `Xdump:tool:events=vmstop,exec=myProgram,opts=WAIT1000`

This option starts the process, waits for the process to end, and then waits a further 1 second (1000 milliseconds) before continuing.

- `Xdump:tool:events=vmstop,exec=myProgram,opts=ASYNC+WAIT10000`

Finally the last example starts the process and waits for 10 seconds before continuing, whether the process is still running or not. This last form is useful if you are starting a process that does not end, but requires time to initialize properly.

For more information about using the dump agent tool option, see “Tool option” on page 326.

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
java -Xdump:heap:defaults -version
```

Default `-Xdump:heap` settings:

```plain
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:

```plain
-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 335.

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:

<table>
<thead>
<tr>
<th>Option value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exclusive</td>
<td>Request exclusive access to the JVM.</td>
</tr>
<tr>
<td>compact</td>
<td>Run garbage collection. This option removes all unreachable objects from the heap before the dump is generated.</td>
</tr>
<tr>
<td>prepwalk</td>
<td>Prepare the heap for walking. You must also specify <code>exclusive</code> when you use this option.</td>
</tr>
<tr>
<td>serial</td>
<td>Suspend other dumps until this dump is finished.</td>
</tr>
<tr>
<td>preempt</td>
<td>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 Java dump.</td>
</tr>
</tbody>
</table>

You can specify more than one request option by using `. For example:

```plain
-Xdump:heap:request=exclusive+compact+prepwalk
```
The JVM exclusive access mechanism allows a JVM thread to halt the activity of other JVM threads in a controlled way by using internal JVM locks. When the `request=exclusive` option is specified for a dump agent, the JVM thread that is producing the dump waits for threads that are running Java code to halt, and for garbage collection operations to complete, before the dump is written. This process helps ensure that the dump has consistent data. When the dump is complete, the mechanism allows the other threads to resume.

By default, only system dumps for OutOfMemoryError exceptions request exclusive access. Other system dump events typically result from a crash. In these cases, exclusive access is not requested because acquiring locks during a crash can be problematic.

If system dumps are requested by using the com.ibm.jvm.Dump.SystemDump() API, the default system dump agent settings are used, and exclusive access is not requested. However, if you intend to use the system dump for Java heap memory analysis, use the following option to request exclusive access when the dump is taken:

```
-Xdump:system:defaults:request=exclusive+compact+prewalk
```

These settings avoid capturing a dump with in-flight data during garbage collection.

As an alternative, you can use the com.ibm.jvm.Dump.triggerDump() API and specify 'request=exclusive+compact+prepwalk' on the API call.

For more information about the com.ibm.jvm.Dump API, see API reference information.

The default setting of the request option for Java dumps is `request=exclusive+preempt`. To change the settings so that Java dumps 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.

### defaults option:

Each dump type has default options. To view the default options for a particular dump type, use `-Xdump:<type>::defaults`.

You can change the default options at run time. For example, you can direct Java dump files into a separate directory for each process, and guarantee unique files by adding a sequence number to the file name using:

```
-Xdump:java:defaults:file=dumps/%pid/javacore-%seq.txt
```

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 directories, and to pass command-line arguments to the tool agent.

The tokens available are listed in the following table:

<table>
<thead>
<tr>
<th>Token</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Y</td>
<td>Year (4 digits)</td>
</tr>
<tr>
<td>%y</td>
<td>Year (2 digits)</td>
</tr>
<tr>
<td>%m</td>
<td>Month (2 digits)</td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month (2 digits)</td>
</tr>
<tr>
<td>%H</td>
<td>Hour (2 digits)</td>
</tr>
<tr>
<td>%M</td>
<td>Minute (2 digits)</td>
</tr>
<tr>
<td>%S</td>
<td>Second (2 digits)</td>
</tr>
<tr>
<td>%pid</td>
<td>Process id</td>
</tr>
<tr>
<td>%uid</td>
<td>User name</td>
</tr>
<tr>
<td>%seq</td>
<td>Dump counter</td>
</tr>
<tr>
<td>%tick</td>
<td>msec counter</td>
</tr>
<tr>
<td>%home</td>
<td>Java home directory</td>
</tr>
<tr>
<td>%last</td>
<td>Last dump</td>
</tr>
</tbody>
</table>

You can use these tokens in the `-Xdump:<agent>:file` command-line option, to modify the dump filename for the specified agent type. You can also use the tokens in the `-Xdump:directory` command-line option, to modify the dump file location for all agent types. For example:

- Organize system dump files by user: `-Xdump:system:file=myApplication-%uid`
- Organize all dump files by user: `-Xdump:directory=/dumps/myApplication-%uid`
- Organize all dump files by process ID: `-Xdump:directory=/dumps/myApplication-%pid`
- Organize all dump files by date: `-Xdump:directory=/dumps/myApplication-%Y%m%d`

**Related information:**

"file option" on page 329

The file option is used by dump agents that write to a file.

**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 337 for more information.

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

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

-Xdump:jit:
  events=gpf+abort,
  label=/home/user/jitdump.%Y%m%d.%H%M%S.%pid.%seq.dmp,
  range=1..0,
  priority=200,
  request=serial
Removing dump agents
You can remove all default dump agents and any preceding dump options by using `-Xdump:none`.

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

-Xdump command-line syntax: the none option

Use this option so that you can subsequently specify a completely new dump configuration.

You can also remove dump agents of a particular type. Here are some examples:

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

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

To turn off all dump agents for corruptcache events:

-`-Xdump:none:events=corruptcache`

To turn off just system dumps for corruptcache events:

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

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

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

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

-`-Xdump:system:none:events=systhrow,filter=java/lang/OutOfMemoryError`
If you remove all dump agents using `-Xdump:none` with no further `-Xdump` options, the JVM still provides these basic diagnostic outputs:

- If a user signal (kill -QUIT) is sent to the JVM, a brief listing of the Java threads including their stacks, status, and monitor information is written to stderr.
- If a crash occurs, information about the location of the crash, JVM options, and native and Java stack traces are written to stderr. A system dump is also written to the user's home directory.

**Tip:** Removing dump agents and specifying a new dump configuration can require a long set of command-line options. To reuse command-line options, save the new dump configuration in a file and use the `-Xoptionsfile` option. See “Specifying command-line options” on page 525 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 ONANY SIGNAL condition is parsed before all others, so

```
ONINTERRUPT(NONE),ONANY SIGNAL(SYSDUMP)
```
has the same effect as
ONANY_SIGNAL(SYS_DUMP), ON_INTERRUPT(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 602.

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 334.
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.

Setting JAVA_DUMP_OPTS only affects those conditions that you specify. Actions on
other conditions are unchanged.

Signal mappings
The signals used in the JAVA_DUMP_OPTS environment variable map to multiple
operating system signals.

When setting the JAVA_DUMP_OPTS environment variable, the mapping of operating
system signals to the “condition” is as follows:

<table>
<thead>
<tr>
<th>Condition</th>
<th>z/OS</th>
<th>Windows</th>
<th>Linux, AIX, and i5/OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCEPTION</td>
<td>SIGTRAP</td>
<td>SIGILL</td>
<td>SIGILL</td>
</tr>
<tr>
<td></td>
<td>SIGILL</td>
<td>SIGSEGV</td>
<td>SISEGV</td>
</tr>
<tr>
<td></td>
<td>SIGFPE</td>
<td>SIGFPE</td>
<td>SIGFPE</td>
</tr>
<tr>
<td></td>
<td>SIGBUS</td>
<td>SIGBUS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIGSYS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIGXFSZ</td>
<td>SIGXFSZ</td>
<td></td>
</tr>
<tr>
<td>INTERRUPT</td>
<td>SIGINT</td>
<td>SIGINT</td>
<td>SIGINT</td>
</tr>
<tr>
<td></td>
<td>SIGTERM</td>
<td>SIGTERM</td>
<td>SIGTERM</td>
</tr>
<tr>
<td></td>
<td>SIGHUP</td>
<td>SIGHUP</td>
<td></td>
</tr>
<tr>
<td>ERROR</td>
<td>SIGABRT</td>
<td>SIGABRT</td>
<td>SIGABRT</td>
</tr>
<tr>
<td>DUMP</td>
<td>SIGQUIT</td>
<td>SIGQUIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIGBREAK</td>
<td></td>
</tr>
</tbody>
</table>
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” 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 363 for more information.
- **JIT dumps**: Output is written to a file named jitdump%Y%m%d.%H%M%S.%pid.%seq.dmp. For more information, see “JIT dumps” on page 328.

Disabling dump agents with -Xrs and -Xrs:sync

When using a debugger such as GDB or WinDbg to diagnose problems in JNI code, you might want to disable the signal handler of the Java runtime environment so that any signals received are handled by the operating system.

Using the -Xrs command-line option prevents the Java runtime environment handling exception signals such SIGSEGV and SIGABRT. When the Java runtime signal handler is disabled, a SIGSEGV or GPF crash does not call the JVM dump agents. Instead, dumps are produced depending on the operating system.

For 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 549.

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 jmdpview 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 255 for more details.

Using Javadump

Javadump produces files that contain diagnostic information that is related to the JVM and a Java application that is captured at a point during execution. For example, the information can be about the operating system, the application environment, threads, stacks, locks, and memory.

Javadumps are human readable and do not contain any Java object content or data, except for the following items:

- Thread names, with thread IDs and flags
- Classloader names, with counts and flags
- Class and method names
- Some heap addresses

The preferred way to control the production of Javadumps is by enabling dump agents using -Xdump:java: on application startup. See “Using dump agents” on page 321. You can also control Javadumps by the use of environment variables. See
“Environment variables and Javadump” on page 362. Default agents are in place that create Javadumps when the JVM ends unexpectedly or when an out-of-memory exception occurs, unless the defaults are overridden. Javadumps are also triggered by default when specific signals are received by the JVM.

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

This chapter describes:
• “Enabling a Javadump”
• “Triggering a Javadump”
• “Interpreting a Javadump” on page 342
• “Environment variables and Javadump” on page 362

Enabling a Javadump
Javadumps are enabled by default. You can turn off the production of Javadumps with -Xdump:java:none.

You are not recommended to turn off Javadumps because they are an essential diagnostic tool.

Use the -Xdump:java option to give more fine-grained control over the production of Javadumps. See “Using dump agents” on page 321 for more information.

Triggering a Javadump
Javadumps can be triggered by error conditions, or can be initiated in a number of ways to obtain diagnostic information.

Javadumps triggered by error conditions

By default, a Javadump is triggered when one of the following error conditions occurs:

An unrecoverable native exception
Not a Java Exception. An “unrecoverable” exception is one that causes the JVM to stop. The JVM handles the event by producing a system dump followed by a snap trace file, a Javadump, and then terminating the process.

The JVM has insufficient memory to continue operation
There are many reasons for running out of memory. See “Problem determination” on page 253 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 com.ibm.jvm.Dump API in your application
The com.ibm.jvm.Dump API provides methods which you can use to trigger a Javadump at run time. Defaults that you set by using the -Xdump option apply to the Javadump that is produced. For information about these defaults, see “Using the -Xdump option” on page 322. You can also set and query JVM dump options by using the API. For more information, see the JVM Dump API.

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 322 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 option” on page 414

**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 between updates to a JVM version. 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 513 for more information.

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

```
1CIJAVAVERSION JRE 1.7.0 AIX ppc64-64 build 20130711_156087 (pap6470_27-20130715_03)
```

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.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0SECTION</td>
<td>TITLE subcomponent dump routine</td>
</tr>
<tr>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>1TICHARSET</td>
<td>ISO8859-1</td>
</tr>
<tr>
<td>1TISIGINFO</td>
<td>Dump Event &quot;vmstart&quot; (00000001) received</td>
</tr>
<tr>
<td>1TIDATETIME</td>
<td>Date: 2014/11/20 at 17:03:57</td>
</tr>
<tr>
<td>1TIFILENAME</td>
<td>Java core filename: /home/user1/javacore.20141120.170357.6226120.0001.txt</td>
</tr>
<tr>
<td>1TIREQFLAGS</td>
<td>Request Flags: 0x81 (exclusive+preempt)</td>
</tr>
<tr>
<td>1TIPREPSTATE</td>
<td>Prep State: 0x106 (vm_access+exclusive_vm_access+)</td>
</tr>
<tr>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>0SECTION</td>
<td>GPINFO subcomponent dump routine</td>
</tr>
<tr>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>2XHOSLEVEL</td>
<td>OS Level: AIX 7.1</td>
</tr>
<tr>
<td>2XHCPUS</td>
<td>Processors:</td>
</tr>
<tr>
<td>3XHCPUARCH</td>
<td>Architecture: ppc64</td>
</tr>
<tr>
<td>3XHNUMCPUS</td>
<td>How Many: 8</td>
</tr>
<tr>
<td>3XHNUMASUP</td>
<td>NUMA is either not supported or has been disabled by user</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0SECTION</td>
<td>TITLE subcomponent dump routine</td>
</tr>
<tr>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>1TICHARSET</td>
<td>ISO8859-1</td>
</tr>
<tr>
<td>1TISIGINFO</td>
<td>Dump Event &quot;gpf&quot; (00002000) received</td>
</tr>
<tr>
<td>1TIDATETIME</td>
<td>Date: 2013/06/20 at 17:03:57</td>
</tr>
<tr>
<td>1TIFILENAME</td>
<td>Java core filename: /home/test/build/javacore.20130620.170357.6226120.0001.txt</td>
</tr>
<tr>
<td>1TIREQFLAGS</td>
<td>Request Flags: 0x81 (exclusive+preempt)</td>
</tr>
<tr>
<td>1TIPREPSTATE</td>
<td>Prep State: 0x100 (trace_disabled)</td>
</tr>
<tr>
<td>1TIPREPINFO</td>
<td>Exclusive VM access not taken: data may not be consistent across javacore sections</td>
</tr>
</tbody>
</table>

**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
The GPINFO section also refers to the vmState, recorded in the console output as VM flags. The vmState is the thread-specific state of what was happening in the JVM at the time of the crash. The value for vmState is a hexadecimal number ending in MSSSS, where M is the SDK component and SSSS is component specific code.

<table>
<thead>
<tr>
<th>SDK component</th>
<th>Code number</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>0x0000</td>
</tr>
<tr>
<td>INTERPRETER</td>
<td>0x1000</td>
</tr>
<tr>
<td>GC</td>
<td>0x2000</td>
</tr>
<tr>
<td>GROW_STACK</td>
<td>0x3000</td>
</tr>
<tr>
<td>JNI</td>
<td>0x4000</td>
</tr>
<tr>
<td>JIT_CODEGEN</td>
<td>0x5000</td>
</tr>
<tr>
<td>BCVERIFY</td>
<td>0x6000</td>
</tr>
</tbody>
</table>
In the example, the value for vmState is VM flags:0000000000000000, which indicates a crash in code outside the SDK. The crash was in the application native library gpf.dll.

When the vmState major component is JIT_CODEGEN, see the information at “JIT and AOT problem determination” on page 430.

**ENVINFO**

Shows information about the Java runtime environment level that failed and details about the command line that launched the JVM process and the JVM environment in place. The information also includes the following lines:

- 1CIJITMODES provides information about JIT settings.
- 1CIPROCESSID shows the ID of the operating system process that produced the core file.

---

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 name of the operating system hypervisor, if any.

The output is similar to the following lines:

---

### Chapter 10. Troubleshooting and support

345
Native memory (NATIVEMEMINFO):

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

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

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

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

Each memory category can have subcategories.

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

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

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

A memory category for Direct Byte Buffers can be found in the VM Class libraries section of the NATIVEMEMINFO output.
You can obtain additional diagnostic information about memory allocation for class library native code. The following output shows the extra information recorded in the Class Libraries section when the system property `-Dcom.ibm.dbgmalloc=true` is set:

```
--Standard Class Libraries: 17,816 bytes / 17 allocations
  +-IO, Math and Language: 1,048 bytes / 1 allocation
  |   +-Zip: 1,048 bytes / 1 allocation
  |   |   +-Wrappers: 3,144 bytes / 3 allocations
  |   |   |   +-Malloc: 1,048 bytes / 1 allocation
  |   |   |   |   +-z/OS EBCDIC Conversion: 1,048 bytes / 1 allocation
  |   |   |   |   |   +-Other: 1,048 bytes / 1 allocation
  |   |   |   |   |   |   +-Networking: 4,192 bytes / 4 allocations
  |   |   |   |   |   |   |   +-NET: 1,048 bytes / 1 allocation
  |   |   |   |   |   |   |   |   +-NIO and NIO.2: 1,048 bytes / 1 allocation
```
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 69 for details about how the Memory Manager works.

This part of the Javadump provides various storage management values in hexadecimal. The information also shows the free memory, used memory and total memory for the heap, in decimal and hexadecimal. If an initial maximum heap size, or soft limit, is specified using the \(-Xsoftmx\) option, this is also shown as the target memory for the heap. For more information about \(-Xsoftmx\), see “Garbage Collector command-line options” on page 577.

This section also contains garbage collection history data, described in “Default memory management tracing” on page 395. Garbage collection history data is shown as a sequence of tracepoints, each with a timestamp, ordered with the most recent tracepoint first.

In the Javadump, segments are blocks of memory allocated by the Java runtime environment for tasks that use large amounts of memory. Example tasks are maintaining JIT caches, and storing Java classes. The Java runtime environment also allocates other native memory, that is not listed in the MEMINFO section. The total memory used by Java runtime segments does not necessarily represent the complete memory footprint of the Java runtime environment. A Java runtime segment consist of the segment data structure, and an associated block of native memory.

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

- Object memory section (HEAPTYPE):

  | id | The ID of the space or region. | RMI: 1,048 bytes / 1 allocation |
  |    |                              | +--RMI: 1,048 bytes / 1 allocation |
  |    |                              | +--GUI: 5,240 bytes / 5 allocations |
  |    |                              | +--AWT: 1,048 bytes / 1 allocation |
  |    |                              | +--MAWT: 1,048 bytes / 1 allocation |
  |    |                              | +--JAWT: 1,048 bytes / 1 allocation |
  |    |                              | +--Medialib Image: 1,048 bytes / 1 allocation |
  |    |                              | +--Other: 1,048 bytes / 1 allocation |
  |    | Font: 1,048 bytes / 1 allocation |
  |    | Sound: 1,048 bytes / 1 allocation |
  |    | Other: 1,048 bytes / 1 allocation |
**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 (SECTYPE), including class memory, JIT code cache, and JIT data cache:

  **segment**

  The address of the segment control data structure.

  **start**

  The start address of the native memory segment.

  **alloc**

  The current allocation address within the native memory segment.

  **end**

  The end address of the native memory segment.

  **type**

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

  **size**

  The size of the native memory segment.
Locks, monitors, and deadlocks (LOCKS):

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

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

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

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

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

This classic deadlock situation is caused by an error in application design; the Javadump tool is a major tool in the detection of such events.
Blocked thread information is also available in the Threads section of the Java dump, in lines that begin with 3XTHREADBLOCK, for threads that are blocked, waiting or parked. For more information, see “Blocked thread information” on page 357.

---

### 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:

```
3XTHREADINFO "main" J9VMThread:0x0000000000F010500, jthread_t:0x00000000113AC30, java/lang/Thread:0x00000000E0030068, state:R, prio=5
3XJAVAVALTHREAD (java/lang/Thread::gettid(), 1, isDaemon:False)
3XTHREADINFO01 (native thread ID:0x8b14, native priority:0x5, native policy:UNKNOWN, vmstate:OC, vm thread flags:0x00000001)
3XMCPUTIME CPU usage total: 0.636904180 secs, user: 0.546904350 secs, system: 0.093000000 secs
3XMEAPALOLOC Heap bytes allocated since last GC cycle=285776 (0x3C810)
3XTHREADINFO03 Java stackcall:
4XSTACKTRACE at myloop.main(myLoop.java:32[Compiled Code])
3XTHREADINFO03 Native callstack:
4XENATIVESTACK ZWaitForSingleObjectEx40x [0x00000000007C452FA [ntdll!0x512fa]]
4XENATIVESTACK WaitForSingleObjectEx40x9c [0x00000000007C452DC [KERNELBASE!0x106c]]
4XENATIVESTACK monitor_wait_original40x3e [j9thread.c:3766, 0x000000007FE4C6000 [j9thr27+0x6500e]]
4XENATIVESTACK j9thread_monitor_wait40x43 [j9thread.c:3492, 0x000000007FE4C6993 [j9thr27+0x6993]]
4XENATIVESTACK internalAcquireMutexWithMask40x3c2 [vmaccess.c:330, 0x000000007FED6E2C [j9vm27+0x6e02c]]
4XENATIVESTACK javaCheckAsyncMessages+0xe9 [async.asm:156, 0x000000007FEB1609 [j9vm27+0x16099]]
```

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

- Native operating system thread ID
- Native operating system thread priority
- Native operating system scheduling policy
- Internal VM thread state
- Internal VM thread flags
The Java thread priority is mapped to an operating system priority value in a platform-dependent manner. A large value for the Java thread priority means that the thread has a high priority. In other words, the thread runs more frequently than lower priority threads.

The values of the Java thread state and the internal VM thread 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:
  - 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 357.

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

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

If the Java dump was triggered by an exception throw, catch, uncaught, or systhrow event, or by the com.ibm.jvm.Dump API, the output contains the stored tracepoint history for the thread. For more information, see “Trace history for the current thread” on page 359.

When you initiate a Java dump to obtain diagnostic information, the JVM quiesces Java threads before producing the javacore. A preparation state of exclusive_vm_access is shown in the 1TIPREPSTATE line of the TITLE section. 1TIPREPSTATE Prep State: 0x4 (exclusive_vm_access)

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

Previous behavior

Before service refresh 2, threads that were running Java code when the javacore was triggered show the thread state as in CW (Condition Wait) state, for example:
The javacore LOCKS section shows that these threads are waiting on an internal JVM lock.

2LKREGMON Thread public flags mutex lock (0x0000000011A8CF8): <unowned>
3LKNOTIFYQ Waiting to be notified:
3LKWAITNOTIFY "main" (J9VMThread:0x000000000F010500)

Understanding Java and native thread details:

There are three types of stack trace: 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 333. 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:

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

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

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

Java locks are released when the Object.wait() method is called. Therefore a record of a thread entering a lock in its stack does not guarantee that the thread still holds the lock. The thread might be waiting to be notified about the lock, or it might be blocked while attempting to re-enter the lock after being notified. In particular, if another thread calls the Object.notifyAll() method, all threads that are waiting for that monitor must compete to re-enter it, and some threads will become blocked. You can determine whether a thread is blocked or waiting on a lock by looking at the 3XMTHREADBLOCK line for that thread. For more information see "Blocked thread information" on page 357. A thread that calls the Object.wait() method releases the lock only for the object that it called the Object.wait() method on. All other locks that the thread entered are still held by that thread.

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

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

### Attached native thread

The attached native threads provide the same set of information as a Java and native thread pair, but do not have a Java stack trace. For example:

```
"JIT Compilation Thread" TID:0x01E92300, j9thread_t:0x00295780, state:CW, prio=10
(native stack ID:0x3030, native priority:0x8, native policy:UNKNOWN)
No Java callstack associated with this thread
Native callstack:
KifastSystemCallRet+0x0  (0x76B2860C) [ntdll+0x2860C]
WaitForSingleObject+0x12  (0x776E1CD0) [kernel32+0x21c8d]
monitor_wait_original+0x5a0 (j9thread.c:3593, 0x7FFA49F0 [J9THR26+0x49F0])
monitor_wait+0x5f  (j9thread.c:3439, 0x7FFA443F [J9THR26+0x443F])
```

---

### 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, 0x77FA24C7 [J9THR26+0x24c7])
  samplerThreadProc+0x261 (hookedbythejit.cpp:3227, 0x00449C21 [j9jit+0x19c21])
  thread_wrapper+0x133 (j9thread.c:975, 0x7FFA24C7 [J9THR26+0x24c7])
  _threadstart+0x6c (thread.c:196, 0x77E6482F [msvcr71+0x2482f])
```

Java dumps are triggered in two distinct ways that influence the structure of the THREADS section:

**A general protection fault (GPF) occurs:**

The Current thread subsection contains only the thread that generated the GPF. The other threads are shown in the Thread Details subsection.

**A user requests a Java dump for an event using, for example, the kill -QUIT command or the com.ibm.jvm.Dump.JavaDump API:**

There is no Current thread subsection and all threads are shown in the Thread Details subsection.

The following example is an extract from the THREADS section that was generated when the main thread caused a GPF.

```null
1XMCURTHDINFO  Current thread
2XMTHREADINFO  Thread Details
```

### Example of THREADS section

<table>
<thead>
<tr>
<th>Section</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>TH1</td>
<td>Main thread details</td>
</tr>
<tr>
<td>TH2</td>
<td>Another thread details</td>
</tr>
</tbody>
</table>

#### Main thread details

```
1XMCURTHDINFO  Current thread
2XMTHREADINFO  Thread Details
```

#### Another thread details

```
1XMCURTHDINFO  Current thread
2XMTHREADINFO  Thread Details
```

---

Chapter 10. Troubleshooting and support 355
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).

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:

```
```
The group IDs are:

- SYMBOL = 1
- SYMBOL_OFFSET = 2
- ID = 3
- IP = 4
- MODULE = 5
- MODULE_OFFSET = 6

**Blocked thread information:**

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

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

- **state:P**
  - Parked threads

- **state:B**
  - Blocked threads

- **state:CW**
  - Waiting threads

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

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

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

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

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

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

Example: a blocked thread

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

```
3XTHREADINFO  "Thread-5" J9VMThread:0x4F6E4100, jthread_t:0x501C0A28, java/lang/Thread:0x4D8C9520, state:B, prio=5
3XTHREADINFO1 (native thread ID:0x664, native priority:8x5, native policy:UNKNOWN)
3XTHREADBLOCK Blocked on: java/lang/String@0x4D8C90F8 Owned by: "main" (J9VMThread:0x00129100, java/lang/Thread:0x000129100)
```

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:

```
3XTHREADINFO  "Thread-5" J9VMThread:0x00503D00, jthread_t:0x00A0ADB8, java/lang/Thread:0x68E04F90, state:CW, prio=5
3XTHREADINFO1 (native thread ID:0x00503D00, native priority:0x5, native policy:UNKNOWN)
3XTHREADBLOCK Waiting on: java/lang/String@0x68E63E60 Owned by: "main" (J9VMThread:0x6B3F9A00, java/lang/Thread:0x6B3F9A00)
```

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: 0x00A0A0D4:
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:

```
3XTHREADINFO  "Thread-5" J9VMThread:0x4F970200, jthread_t:0x501C0A28, java/lang/Thread:0x4D9AD640, state:P, prio=5
3XTHREADINFO1 (native thread ID:0x157, native priority:0x5, native policy:UNKNOWN)
3XTHREADBLOCK Parked on: java/util/concurrent/locks/ReentrantLock$NonfairSync@0x4D9ACCF0 Owned by: "main" (J9VMThread:0x00129100, java/lang/Thread:0x000129100)
```

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:
In this example, the thread that is holding the lock, Thread-5, ended without using the unlock() method to release the lock. 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:

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:

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

Trace history for the current thread:

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

For Java dumps that were triggered by exception throw, catch, uncaught, and systhrow events (see “Dump events” on page 328 for more information) or by the com.ibm.jvm.Dump API, extra lines are output at the end of the THREADS section. These lines show the stored tracepoint history for the thread, with the most recent tracepoints first. The trace data is introduced by the following line:

```
1XECTHTYPE  Current thread history (J9VMThread:0x00000000002BA0500)
```

The tracepoints provide detailed information about the JVM, JIT, or class library code that was run on the thread immediately before the Java dump was triggered. In the following example, the Java dump was triggered by a java/lang/VerifyError exception. The tracepoints show that the reason for the exception was that a method in a class was overridden, but is defined as final in a superclass (see tracepoint J9vm.91 in the example). The output also shows the names of the classes that were being loaded by the JVM when the exception occurred.
Note: The timestamp information is replaced by (time) to reduce the amount of information that needs to be shown.

3XEHSTTYPE (time) j9dmp.9 - Preparing for dump, filename=c:\test\verifyerrorjavacore.20140124.155651.4544.0001.txt
3XEHSTTYPE (time) j9v.2 - <Created RAM class 0000000000000000 from ROM class 0000000000000000
3XEHSTTYPE (time) j9v.304 - setCurrentExceptionUTF
3XEHSTTYPE (time) j9v.301 - setCurrentException
3XEHSTTYPE (time) j9v.5 - exitInterpreter
3XEHSTTYPE (time) j9v.10 - >internalSendExceptionConstructor
3XEHSTTYPE (time) j9v.353 - <loader 0000000002C19000 class 000000000002440FAB attemptDynamicClassLoad exit
3XEHSTTYPE (time) j9v.2 - <Created ROM class 0000000002C19000 from ROM class 0000000000000000
3XEHSTTYPE (time) j9v.80 - ROM class 0000000000000000 is named java/lang/VerifyError
3XEHSTTYPE (time) j9v.1 - <Create RAM class from ROM class 0000000000000000 in class loader 0000000000000000
3XEHSTTYPE (time) j9v.351 - >loader 0000000000000000 class java/lang/VerifyError attemptDynamicClassLoad entry
3XEHSTTYPE (time) j9v.248 - <dispatchAsyncEvents
3XEHSTTYPE (time) j9v.247 - call event handler: handlerKey=0 eventHandler=000007FEEE79AF20 userData=0000000000000000
3XEHSTTYPE (time) j9v.246 - >dispatchAsyncEvents asyncEventFlags=0000000000000000
3XEHSTTYPE (time) j9v.119 - send loadClass(java/lang/VerifyError), stringObject: 0000000000000000 loader: 0000000000000000
3XEHSTTYPE (time) j9v.316 - 0X00000000000000000000000000004CE448 class java/lang/VerifyError attemptDynamicClassLoad exit
3XEHSTTYPE (time) j9v.318 - 0X00000000000000000000000000004CE448 class java/lang/VerifyError overridden
3XEHSTTYPE (time) j9v.317 - 0X00000000000000000000000000004CE448 class java/lang/VerifyError attemptDynamicClassLoad exit
3XEHSTTYPE (time) j9v.315 - 0X00000000000000000000000000004CE448 class java/lang/VerifyError attemptDynamicClassLoad exit
3XEHSTTYPE (time) j9v.294 - >setCurrentException index=55 constructorIndex=0 detailMessage=0000000000000000
3XEHSTTYPE (time) j9v.302 - >setCurrentExceptionUTF exceptionNumber=55 detailUTF=JVMVRFY007 final method overridden; class=B, method=bad()V
3XEHSTTYPE (time) j9v.91 - Method bad()V is overridden, but it is final in a superclass. Throw VerifyError
3XEHSTTYPE (time) j9v.319 - 0X00000000000000000000000000004CE5F8 class A arbitratedLoadClass exit foundClass 0000000000000000
3XEHSTTYPE (time) j9v.120 - sent loadClass(A) -- got 0000000000000000
3XEHSTTYPE (time) j9v.2 - <Created RAM class 00000000000000002C09600 from ROM class 0000000000000000
3XEHSTTYPE (time) j9v.80 - ROM class 0000000000000000 is named A
3XEHSTTYPE (time) j9v.1 - <Create RAM class from ROM class 0000000000000000 in class loader 0000000000000000
3XEHSTTYPE (time) j9v.80 - ROM class 0000000000000000 is named A
3XEHSTTYPE (time) j9v.353 - <loader 00000000000000000000000000004CE5F8 attemptDynamicClassLoad exit
3XEHSTTYPE (time) j9v.304 - >dispatchAsyncEvents asyncEventFlags=0000000000000000
3XEHSTTYPE (time) j9v.319 - 0X00000000000000000000000000004CE5F8 class A arbitratedLoadClass exit
3XEHSTTYPE (time) j9v.317 - 0X00000000000000000000000000004CE5F8 class A attemptDynamicClassLoad entry
3XEHSTTYPE (time) j9v.315 - >loader 00000000000000000000000000004CE5F8 class A attemptDynamicClassLoad exit
3XEHSTTYPE (time) j9v.319 - 0X00000000000000000000000000004CE5F8 class A arbitratedLoadClass exit
3XEHSTTYPE (time) j9v.317 - 0X00000000000000000000000000004CE5F8 class A attemptDynamicClassLoad entry
3XEHSTTYPE (time) j9v.315 - >loader 00000000000000000000000000004CE5F8 class A attemptDynamicClassLoad exit

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 482 for a description of the summary information.

SHARED CLASSES subcomponent dump routine

Cache Created With

- XnolineNumbers = false
- BCI Enabled = true

Cache Summary

No line number content = false
Line number content = false

ROMClass start address = 0x629EC000
ROMClass end address = 0x62AD468
Metadata start address = 0x636F9800
Cache end address = 0x639D000
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 101 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.

Environment variables and Javadoc

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 Javadoc production:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Usage Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISABLE_JAVADUMP</td>
<td>Setting DISABLE_JAVADUMP to true is the equivalent of using -Xdump:java:none and stops the default production of javadumps.</td>
</tr>
<tr>
<td>IBM_JAVACOREDIR</td>
<td>The default location into which the Javacore will be written.</td>
</tr>
</tbody>
</table>
### Environment Variable

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Usage Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA_DUMP_OPTS</td>
<td>Use this environment variable to control the conditions under which Javadumps (and other dumps) are produced. See “Dump agent environment variables” on page 338 for more information.</td>
</tr>
<tr>
<td>IBM_JAVADUMP_OUTOFMEMORY</td>
<td>By setting this environment variable to false, you disable Javadumps for an out-of-memory exception.</td>
</tr>
</tbody>
</table>

### Using Heapdump

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

There are two Heapdump formats, the text or classic Heapdump format and the Portable Heap Dump (PHD) format. Although the text or classic format is human-readable, the PHD format is compressed and is not human-readable. Both Heapdump formats contain a list of all object instances in the heap, including each object address, type or class name, size, and references to other objects. The Heapdumps also contain information about the version of the JVM that produced the Heapdump. They do not contain any object content or data other than the class names and the values (addresses) of the references.

You can use various tools on the Heapdump output to analyze the composition of the objects on the heap. This analysis might help to find the objects that are controlling large amounts of memory on the Java heap and determine why the Garbage Collector cannot collect them.

This chapter describes:

- “Getting Heapdumps”
- “Available tools for processing Heapdumps” on page 364
- “Using -Xverbose:gc to obtain heap information” on page 364
- “Environment variables and Heapdump” on page 364
- “Text (classic) Heapdump file format” on page 365
- “Portable Heap Dump (PHD) file format” on page 367

### Getting Heapdumps

By default, a Heapdump is produced when the Java heap is exhausted. Heapdumps can be generated in other situations by use of -Xdump:heap.

To see which events will trigger a dump, use -Xdump:what. See “Using dump agents” on page 321 for more information about the -Xdump parameter.

You can also use the com.ibm.jvm.Dump.HeapDump() method in your application code, to generate a Heapdump programmatically. Other methods are also available in the com.ibm.jvm.Dump API. For more information, see JVM Dump API.

By default, Heapdumps are produced in PHD format. To produce Heapdumps in text format, see “Enabling text formatted ("classic") Heapdumps” on page 364.
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 321). 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 websites.

The preferred Heapdump analysis tool is the IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer. The tool is available in IBM Support Assistant: [https://www.ibm.com/software/support/isa/](https://www.ibm.com/software/support/isa/) Information about the tool can be found at [https://www.ibm.com/developerworks/java/jdk/tools/memoryanalyzer/](https://www.ibm.com/developerworks/java/jdk/tools/memoryanalyzer/)

Further details of the range of available tools can be found at [https://www.ibm.com/support/docview.wss?uid=swg24009436](https://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 69.

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:
<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Usage Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM_HEAPDUMP</td>
<td>Setting either of these to any value (such as true) enables heap dump production by</td>
</tr>
<tr>
<td>IBM_HEAP_DUMP</td>
<td>means of signals.</td>
</tr>
<tr>
<td>IBM_HEAPDUMPDIR</td>
<td>The default location into which the</td>
</tr>
<tr>
<td></td>
<td>Heapdump will be written.</td>
</tr>
<tr>
<td>JAVA_DUMP_OPTS</td>
<td>Use this environment variable to control the conditions under which Heapdumps (and</td>
</tr>
<tr>
<td></td>
<td>other dumps) are produced. See &quot;Dump agent environment variables&quot; on page 338</td>
</tr>
<tr>
<td></td>
<td>for more information.</td>
</tr>
<tr>
<td>IBM_HEAPDUMP_OUTOFMEMORY</td>
<td>By setting this environment variable to false, you disable Heapdumps for an</td>
</tr>
<tr>
<td></td>
<td>OutOfMemory condition.</td>
</tr>
<tr>
<td>IBM_JAVA_HEAPDUMP_TEST</td>
<td>Use this environment variable to cause the JVM to generate both phd and text</td>
</tr>
<tr>
<td></td>
<td>versions of Heapdumps. Equivalent to opts=PHD+CLASSIC on the -Xdump:heap option.</td>
</tr>
<tr>
<td>IBM_JAVA_HEAPDUMP_TEXT</td>
<td>Use this environment variable to cause the JVM to generate a text (human readable)</td>
</tr>
<tr>
<td></td>
<td>Heapdump. Equivalent to opts=CLASSIC on the -Xdump:heap option.</td>
</tr>
</tbody>
</table>

**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: JRE 1.7.0 AIX ppc64-64 build 20120817_119700 (pap6470sr3-20120821_01(SR3))
```

**Object records**

Object records are multiple records, one for each object instance on the heap, providing object address, size, type, and references from the object.

```
<object address, in hexadecimal> [<length in bytes of object instance, in decimal>] OBJ <object type>
<heap reference, in hexadecimal> <heap reference, in hexadecimal> ...
```

The object type is either a class name, or a class array type, or a primitive array type, shown by the standard JVM type signature, see "Java VM type signatures" on page 366. Package names are included in the class names. References found in the object instance are listed, excluding references to an object's class and excluding null references.

Examples:

An object instance, length 32 bytes, of type java/lang/String, with its single reference to a char array:
An object instance, length 72 bytes, of type char array, as referenced from the java/lang/String:
0x000007FFFFFF842F0 [72] OBJ [C

An object instance, length 48 bytes, of type array of java/lang/String
0x000007FFFFFF84C8B [48] OBJ [Ljava/lang/String;

Class records

Class records are multiple records, one for each loaded class, providing class block
address, size, type, and references from the class.

The class type is either a class name, or a class array type, or a primitive array
type, shown by its standard JVM type signature, see "Java VM type signatures." Package names are included in the class names. References found in the class block
are listed, excluding null references.

Examples:

A class object, length 80 bytes, for class java/util/Vector, with heap references:
0x000007FFDFFC2F80 [80] CLS java/util/Vector

Trailer record 1

Trailer record 1 is a single record containing record counts.

For example:

Trailer record 2

Trailer record 2 is a single record containing totals.

For example:

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

<table>
<thead>
<tr>
<th>Java type</th>
<th>Java VM type signatures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>boolean</td>
</tr>
<tr>
<td>B</td>
<td>byte</td>
</tr>
<tr>
<td>C</td>
<td>char</td>
</tr>
<tr>
<td>S</td>
<td>short</td>
</tr>
<tr>
<td>I</td>
<td>int</td>
</tr>
<tr>
<td>J</td>
<td>long</td>
</tr>
<tr>
<td>F</td>
<td>float</td>
</tr>
<tr>
<td>D</td>
<td>double</td>
</tr>
<tr>
<td>L</td>
<td>&lt;fully qualified-class&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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.
  - Object array record. Indicated by having a tag value of 5.
  - Class record. Indicated by having a tag value of 6.
  - Object array record (revised). Indicated by having a tag value of 8.

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

Different versions of PHD are produced, depending on the version of the J9 virtual machine (VM):
- J9 VM version 2.7 produces PHD version 6.
- J9 VM versions 2.4, 2.5, and 2.6 produce PHD version 5.

To find out which IBM J9 VM you are using with the IBM SDK or Java runtime environment, type java -version on the command line and inspect the output.

**PHD object records:**

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

**Short object record**

The short object record includes detailed information within the tag “byte”. This information includes:
- The 1 byte tag. The top bit (0x80) is set and the following 7 bits in descending order contain:
  - 2 bits for the class cache index. The value represents an index into a cache of the last four classes used.
  - 2 bits containing the number of references. Most objects contain 0 - 3 references. If there are 4 - 7 references, the medium object record is used. If there are more than seven references, the long object record is used.
1 bit to indicate whether the gap is a “byte” or a “short”. The gap is the difference between the address of this object and the previous object. If set, the gap is a “short”. If the gap does not fit into a “short”, the “long” object record form is used.

2 bits indicating the size of each reference. The following values apply:
- 0 indicates “byte” format.
- 1 indicates “short” format.
- 2 indicates “integer” format.
- 3 indicates “long” format.

- A “byte” or a “short” containing the gap between the address of this object and the address of the preceding object. The value is signed and represents the number of 32-bit “words” between the two addresses. Most gaps fit into 1 byte.
- If all objects are hashed, a “short” containing the hashcode.
- The array of references, if references exist. The tag shows the number of elements, and the size of each element. The value in each element is the gap between the address of the references and the address of the current object. The value is a signed number of 32-bit “words”. Null references are not included.

Medium object record

These records provide the actual address of the class rather than a cache index. The format is:
- The 1 byte tag. The second bit (0x40) is set and the following 6 bits in descending order contain:
  - 3 bits containing the number of references.
  - 1 bit to indicate whether the gap is a 1 byte value or a “short”. For more information, see the description in the short record format.
  - 2 bits indicating the size of each reference. For more information, see the description in the short record format.
- A “byte” or a “short” containing the gap between the address of this object and the address of the preceding object. For more information, see the description in the short record format.
- A “word” containing the address of the class of this object.
- If all objects are hashed, a “short” containing the hashcode.
- The array of references. For more information, see the description in the short record format.

Long object record

This record format is used when there are more than seven references, or if there are extra flags or a hashcode. The record format is always used for packed objects. 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.
  - 1 bit indicating if the object is packed (PHD V6 and later).
  - 1 bit indicating if the object is native packed (PHD V6 and later).
- 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.
- If the class is packed (0x08 bit in the flags “byte”), an unsigned “int” containing the size of the instance of the object on the heap, including header and padding. The size is measured in 32-bit words, which you can multiply by four to obtain the size in bytes. This format allows encoding of lengths up to 16GB in an unsigned “int” (PHD V6 and later).

**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.
- An unsigned “int” containing the size of the instance of the array on the heap, including header and padding. The size is measured in 32-bit words, which you
can multiply by four to obtain the size in bytes. This format allows encoding of lengths up to 16GB in an unsigned “int”. (PHD V6 and later)

**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.
- An unsigned “int” containing the size of the instance of the array on the heap, including header and padding. The size is measured in 32-bit words, which you can multiply by four to obtain the size in bytes. This format allows encoding of lengths up to 16GB in an unsigned “int”. (PHD V6 and later)

**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.
  - 1 bit indicating if the object is packed (PHD V6 and later).
  - 1 bit indicating if the object is native packed (PHD V6 and later).
  - 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.
• An unsigned “int” containing the size of the instance of the array on the heap, including header and padding. The size is measured in 32-bit words, which you can multiply by four to obtain the size in bytes. This format allows encoding of lengths up to 16GB in an unsigned “int” (PHD V6 and later).

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.
  – 1 bit indicating if the class is packed (PHD V6 and later).
• A “byte”, “short”, “int” or “long” containing the gap between the address of this class and the address of the preceding object. For more information, see the description in the short record format.
• An “int” containing the instance size.
• If all objects are hashed, a “short” containing the hashcode. Otherwise, an optional “int” containing the hashcode if the hashed and moved bit is set in the record flag byte.
• A “word” containing the address of the superclass.
• A UTF string containing the name of this class.
• An “int” containing the number of static references.
• The array of static references. For more information, see the description in the short record format.
Using system dumps and the dump viewer

The JVM can generate native system dumps, also known as core dumps, under configurable conditions.

System dumps contain a binary copy of process memory and are not human-readable. The dumps contain a complete copy of the Java heap, including the contents of all Java objects in the application. If you require a dump that does not contain this application data, see these topics: “Using Javadoc” on page 340 or “Using Heapdump” on page 363.

Dump agents are the preferred method for controlling the generation of system dumps. For more information, see “Using dump agents” on page 321. To maintain compatibility with earlier versions, the JVM supports the use of environment variables for triggering system dumps. For more information, see “Dump agent environment variables” on page 338.

The dump viewer that is provided in the SDK is a cross-platform line-mode tool for viewing and analyzing system dumps. The following operating system tools can also be used:

- AIX: dbx
- Linux: gdb
- Windows: windbg
- z/OS: ISPF

This chapter tells you about system dumps and how to use the dump viewer. It contains these topics:

- “Overview of system dumps”
- “System dump defaults” on page 374
- “Using the dump viewer” on page 374

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=7FA8506D ContextFlags=0001003f Handler1=7FED79C0 Handler2=7FED8C00 InaccessibleAddress=0000001C EDI=41FEC3F0 ESI=00000000 EAX=41F80E60 EBX=41EE6C01 ECX=41C5F9C0 EDX=41F80E60 EIP=7FA8506D ESP=41C5F940 EBP=41EE6CA4 Module=E:\testjava\java7-32\sdk\jre\bin\j9jit24.dll Module_base_address=7FB00000 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 321 for more information on dump agents.

**System dump defaults**

There are default agents for producing system dumps when using the JVM.

Using the `-Xdump:what` option shows the following system dump agent:

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

This output shows that by default a system dump is produced in these cases:

- A general protection fault occurs. (For example, branching to memory location 0, or a protection exception.)
- An abort is encountered. (For example, native code has called abort() or when using kill -ABRT on Linux)

**Attention:** The JVM used to produce this output when a SIGSEGV signal was encountered. This behavior is no longer supported. Use the ABRT signal to produce dumps.

**Using the dump viewer**

System dumps are produced in a platform-specific binary format, typically as a raw memory image of the process that was running at the time the dump was initiated. The SDK dump viewer allows you to navigate around the dump, and obtain information in a readable form, with symbolic (source code) data where possible.

You can view Java information (for example, threads and objects on the heap) and native information (for example, native stacks, libraries, and raw memory locations). You can run the dump viewer on one platform to work with dumps from another platform. For example, you can look at Linux dumps on a Windows platform.

You can also explore the dump file by using IBM Monitoring and Diagnostic Tools - Interactive Diagnostic Data Explorer. This tool is a GUI-based version of the dump viewer, which provides extra functionality such as command assistance and the ability to save the tool output.

**Dump viewer: jdmpview**

The dump viewer is a command-line tool that allows you to examine the contents of system dumps produced from the JVM. The dump viewer requires metadata created by the `jextract` utility, if the system dump was generated by a version of the IBM J9 virtual machine before V2.6. To check the version of a JVM, use the `java -version` command and examine the output. The dump viewer allows you to view both Java and native information from the time the dump was produced.

`jdmpview` is in the directory `sdk/bin`.

To start `jdmpview`, from a shell prompt, enter:

```
jdmpview -zip <zip file>
```
The `jdmpview` tool accepts these parameters:

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

- **-notemp**
  
  By default, when you specify a file by using the `-zip` option, the contents are extracted to a temporary directory before processing. Use the `-notemp` option to prevent this extraction step, and run all subsequent commands in memory.

- **-xml <xml file>**
  
  Specify a metadata file. `jdmpview` guesses the name of the XML file if the `-xml` option is not present. This option is not required for core files generated from an IBM J9 2.6 virtual machine.

- **-zip <zip file>**
  
  Specify a compressed file containing the core file and associated XML file (produced by `jextract`).

**Note:** The `-core` and `-xml` options can be used with the `-zip` option to specify the core and XML files in the compressed file. Without the `-core` or `-xml` options, `jdmpview` shows multiple contexts, one for each source file that it identified in the compressed file. For more information, see “Support for compressed files” on page 376.

After `jdmpview` processes the arguments with which it was started, it shows this message:

For a list of commands, type "help"; for how to use "help", type "help help".

When you see this message, you can start using commands.

When `jdmpview` is used with the `-zip` option, temporary disk space is required to uncompress the dump files from the compressed file. `jdmpview` uses the system temporary directory, `/tmp` on AIX, Linux, or zOS. An alternative temporary directory can be specified using the Java system property `java.io.tmpdir`. `jdmpview` shows an error message if insufficient disk space is available in the temporary directory. Use the `-notemp` option to prevent `jdmpview` from creating these temporary files. The temporary files are deleted when `jdmpview` exits or when you enter the `close` command on the command line.

You can significantly improve the performance of `jdmpview` against large dumps by ensuring that your system has enough memory available to avoid paging. On large dumps (that is, ones with large numbers of objects on the heap), you might have to run `jdmpview` using the `-Xmx` option to increase the maximum heap available. You might also have to increase the maximum heap size if you use the `-notemp` option, especially if you are analyzing a large heap, because this option specifies that all analysis is done in memory.

```
jdmpview -J-Xmx<n> -zip <zip file>
```

To pass command-line arguments to the JVM, you must prefix them with `-J`.

```
jdmpview -core <core file> [-xml <xml file>]
```
Support for compressed files:

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

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

Example 1

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

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

Source: file://C:/test/core.zip#core.20130720.165054.4176.0001.dmp

0 : PID: 4176 : JRE 1.7.0 Windows 7 amd64-64 build 20130711_156087 (pwa6470_27-20130711_01)
```

Example 2

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

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

Source: file:///D:/examples/MV2C.SVCDUMP.D120228.T153207.S00053

0 : ASID: 0x1 : No JRE : No JRE
1 : ASID: 0x3 : No JRE : No JRE
2 : ASID: 0x4 : No JRE : No JRE
3 : ASID: 0x6 : No JRE : No JRE
4 : ASID: 0x7 : No JRE : No JRE
5 : ASID: 0x73 EDB: 0x8004053a0 : JRE 1.7.0 z/OS s390x-64 build 20120228_104045 (pmz6470_27-20120302_01)
6 : ASID: 0x73 EDB: 0x83d2053a0 : JRE 1.7.0 z/OS s390x-64 build 20120228_104045 (pmz6470_27-20120302_01)
7 : ASID: 0x73 EDB: 0xa7b9e8 : No JRE
8 : ASID: 0xffffffff : No JRE : No JRE
```

Example 3

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

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

Source: file:///D:/Samples/multi-image.zip#core1.dmp

0 : PID: 10463 : JRE 1.7.0 Linux amd64-64 build 20120228_104045 pxla6470_27-20120302_01)

Source: file:///D:/Samples/multi-image.zip#core2.dmp

1 : PID: 12268 : JRE 1.7.0 Linux amd64-64 build 20120228_104045 pxla6470_27-20120302_01)

Source: file:///D:/Samples/multi-image.zip#javacore.20120228.100128.10441.0002.txt

2 : JRE 1.7.0 Linux amd64-64 build 20120228_94967 (pxla6470_27-20120228_01)

Source: file:///D:/Samples/multi-image.zip#javacore.20120228.090916.14653.0002.txt

3 : JRE 1.7.0 Linux amd64-64 build 20120228_94967 (pxla6470_27-20120228_01)

Source: file:///D:/Samples/multi-image.zip#heapdump.20130711.093819.4336.0001.phd

4 : JRE 1.7.0 Windows 7 amd64-64 build 20130711_156087 (pwa6470_27-20111107_01)
```
**Working with Java dump and heap dump files**

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

Source: file:///D:/Samples/multi-image.zip#heapdump.20130711.093819.4336.0001.phd

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

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

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

```
CTX:4> info class
instances total size on heap Packed? class name
0 0 sun/io/Converters
1 16 com.ibm/tools/attach/javaSE/FileLock$syncObject
1 32 com.ibm/tools/attach/javaSE/AttachHandler$syncObject
1 40 sun/nio/cs/UTF_16LE
...
```

Total number of objects: 6178
Total size of objects: 2505382

**Support for packed objects:**

The `jdmpview` tool can show you information about packed objects in dump files, such as the type of the packed object, and the references contained within it.

Information about packed objects is produced by the `info class`, `info class` `[<classname>]`, and `x/J` commands.

**Example: info class**

The command output shows if a class is a packed class. The size shown is the total memory that is allocated on the Java heap; this size does not include the native memory that is allocated for off-heap packed objects:

```
> info class
instances total size on heap Packed? class name
...
0 0 sun/util/locale/provider/TimeZoneNameUtility
19 912 java/util/concurrent/ConcurrentHashMap
1 32 com/ibm/dtfj/tests/junit/JavaClassTest
1 16 com/ibm/dtfj/tck/tests/process/image/TestImageGetProcessorSubType
1 72 java/time/temporal/ChronoUnit$Array
0 0 yes com/ibm/dtfj/tests/junit/packed/NestedPackedMixed2
0 0 org/apache/xerces/jaxp/DocumentBuilderImpl
0 0 sun/util/locale/LocaleObjectCache
0 0 java/util/Hashtable$Enumerator
3 688 yes com/ibm/dtfj/tests/junit/packed/PackedPrimitives$Array
...
```
Example: info class [<classname>]

The command output shows if a class is a packed class. The total size of instances shown is the total memory that is allocated on the heap; this size does not include the native memory that is allocated for off-heap packed objects:

> info class com/ibm/dtfj/tests/junit/packed/PackedPrimitives
name = com/ibm/dtfj/tests/junit/packed/PackedPrimitives
ID = 0x24e88f00  superID = 0x24463800
classLoader = 0x2835d90  modifiers: public final

This is a packed class

number of instances: 7
total size of instances on the heap: 176 bytes

Inheritance chain....
com/ibm/jvm/packed/PackedObject
  com/ibm/dtfj/tests/junit/packed/PackedPrimitives

Fields......

static fields for "com/ibm/dtfj/tests/junit/packed/PackedPrimitives"
  public static int staticIntField = 99 (0x63)
  public static long staticLongField = 101 (0x65)

non-static fields for "com/ibm/dtfj/tests/junit/packed/PackedPrimitives"
  public byte byteField
  public boolean booleanField
  public char charField
  public int intField
  public float floatField
  public double doubleField
  public long longField
  public int int2Field

Examples: the x/J command

For packed objects, the x/J command shows information such as the type of the packed object, the size that the object occupies on the heap, the size of the packed data, and the references from the packed object to other objects. A packed object can be one of the following types:

On-heap

An on-heap packed object has data that is stored within the Java heap.

The following example shows that this object is an on-heap object. The references section shows that this object has one reference, to itself. This result is expected because the header of on-heap packed objects contains one self-reference. The references section is most useful for packed objects that contain many references, such as an array of packed objects where each object contains one or more references. For more information, see later examples.

> x/j 0x2941c98
heap #1 - name: Generational@2d3678

com/ibm/dtfj/tests/junit/packed/PackedPrimitives @ 0x2941c98
This is an on-heap packed object occupying 48 bytes on the heap
fields inherited from "com/ibm/jvm/packed/PackedObject":
declared fields:
  public byte byteField = 97 (0x61)
  public boolean booleanField = true
public char charField = '\377' (0xff)
public int intField = 99 (0x63)
public float floatField = 123.4 (0x42f6cccd)
public double doubleField = 123.4 (0x405ed999999999a)
public long longField = 101 (0x65)
public int int2Field = 99 (0x63)

references:
0x2941c98

Off-heap
An off-heap packed object has data that is stored outside the Java heap. This type of packed object is also known as a native packed object.

In the following example, the off-heap packed object occupies fewer bytes (16) on the heap compared to the on-heap packed object of the same class in the previous example. The smaller size is because only an object header is allocated on the heap.

The output for the off-heap packed object also includes the address of the off-heap data and the size of the packed data. The references section shows that there are no references from this object, unlike the on-heap packed object of the same class.

> x/j 0x2941cc8
heap #1 - name: Generational02d3678

com/ibm/dtfj/tests/junit/packed/PackedPrimitives @ 0x2941cc8
This is an off-heap packed object occupying 16 bytes on the heap
The native memory is allocated at 0x24c182f8
The packed data is 32 bytes long
fields inherited from "com/ibm/jvm/packed/PackedObject":
declared fields:
public byte byteField = 0 (0x0)
public boolean booleanField = false
public char charField = '\000' (0x0)
public int intField = 0 (0x0)
public float floatField = 0.0 (0x0)
public double doubleField = 0.0 (0x0)
public long longField = 0 (0x0)
public int int2Field = 0 (0x0)

references: <none>

Derived
A derived packed object is an object header that refers to a nested field within another object.

The jdmpview tool examines the contents of the object header, to show the address of that object and the offset within that object at which the nested field can be found. The tool then follows the reference and examines the fields within the nested packed object. The line beginning showing fields for shows that the tool is examining a nested packed object. In this example, the target object is an array starting at address 0x2941d58, and the derived object refers to the first element of the array, which starts at address 0x2941d68.

> x/j 0x2941ea8
heap #1 - name: Generational02d3678

com/ibm/dtfj/tests/junit/packed/PackedPrimitives @ 0x2941ea8
This is an on-heap packed object occupying 16 bytes on the heap
This is a derived packed object:
target object: com/ibm/dtfj/tests/junit/packed/PackedPrimitives$Array @ 0x2941d58
target offset: 0x10
showing fields for <nested packed object> com/ibm/dtfj/tests/junit/packed/PackedPrimitives packed
fields inherited from "com/ibm/jvm/packed/PackedObject":
declared fields:
public byte byteField = 0 (0x0)
public boolean booleanField = false
public char charField = '\000' (0x0)
public int intField = 0 (0x0)
public float floatField = 0.0 (0x0)
public double doubleField = 0.0 (0x0)
public long longField = 0 (0x0)
public int int2Field = 0 (0x0)

references:
0x2941d58

Nested

A nested packed object is a nested field within a packed object.

Because nested packed objects are packed within a containing object, they do not have an object header, and it is not possible to identify their class from the address alone. To examine a nested packed object, you must therefore supply the class name, the string packed @, and the address. Nested packed objects are displayed in this format in the output of the x/J command, so you can copy output from one command and use it as the input for another command. The following example shows output for a packed field at address 0x2941d68.

> x/j com/ibm/dtfj/tests/junit/packed/PackedPrimitives packed @ 0x2941d68
fields inherited from "com/ibm/jvm/packed/PackedObject":
declared fields:
public int intField = 99 (0x63)
public com.ibm.dtfj.tests.junit.packed.NestedPacked1 nestedPacked1Field = <nested packed object> com/ibm/dtfj/tests/junit/packed/NestedPacked1 packed @ 0x2941240

In the following example, the NestedPacked class contains a nested packed field of type NestedPacked1, which in turn contains a nested packed field of type NestedPacked2. The output from one command is used to provide input to the following command, to explore the nested objects:

> x/j 0x2941210
heap #1 - name: Generational02d3678

com/ibm/dtfj/tests/junit/packed/NestedPacked @ 0x2941210
This is an on-heap packed object occupying 112 bytes on the heap
fields inherited from "com/ibm/jvm/packed/PackedObject":
declared fields:
public int intField = 99 (0x63)
public com.ibm.dtfj.tests.junit.packed.NestedPacked1 nestedPacked1Field = <nested packed object> com/ibm/dtfj/tests/junit/packed/NestedPacked1 packed @ 0x2941240

references:
0x2941d58

> x/j com/ibm/dtfj/tests/junit/packed/NestedPacked1 packed @ 0x2941240
fields inherited from "com/ibm/jvm/packed/PackedObject":
declared fields:
public int intField = 98 (0x62)
public com.ibm.dtfj.tests.junit.packed.NestedPacked2 nestedPacked2Field = <nested packed object> com/ibm/dtfj/tests/junit/packed/NestedPacked2 packed @ 0x2941244
Nested packed objects can also be elements of an array. The following example shows an array of PackedPrimitives objects, containing 10 elements. The example shows how to examine the first element by copying some of the output of the first command to use as a parameter in the second command.

The following example illustrates the use of the references section at the end of the command output. This example uses the PackedMixedTwoRefs class, which is a mixed packed object because it contains references to non-packed Java objects. Because each PackedMixedTwoRefs object contains two references, a packed array of ten such objects contains 20 references to other objects, even though the array is only ten elements long. Here is the output for such an array from a system dump. The 20 references for the packed object are shown in the references section. You can also see the references by running the x/J command on each element in turn.

The example shows an array of PackedPrimitives objects, containing 10 elements. The example shows how to examine the first element by copying some of the output of the first command to use as a parameter in the second command.
Here is the output for the same array from a portable heapdump. The references are shown but the elements are not, because the data within the array is not present in the portable heapdump. You therefore cannot examine the elements within the array any further. The jdumpview tool uses the DTFJ getReferences() API method to get the list of references.

```plaintext
x/j com/ibm/dump/tests/types/packed/PackedMixedTwoRefs$Array
heap #1 - name: Java heap

com/ibm/dump/tests/types/packed/PackedMixedTwoRefs$Array 0x2ef4a690
This is an on-heap packed object occupying 136 bytes on the heap
This is an array of size 10 elements
references:
0x2ef4a690 0x2ef4a748 0x2ef4a718 0x2ef4a7a8 0x2ef4a778 0x2ef4a808
0x2ef4a7d8 0x2ef4a868 0x2ef4a838 0x2ef4a8c8 0x2ef4a898 0x2ef4a928
0x2ef4a8f8 0x2ef4a988 0x2ef4a958 0x2ef4a9e8 0x2ef4a9b8 0x2ef4aa48
0x2ef4aa18 0x2ef4aa88 0x2ef4aa78
```

**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 e8801ed7d21c6be, 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 73 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.

**Using the dump viewer in batch mode:**

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

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

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

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

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

Batch mode is controlled with the following command line options:

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

- **-charset <character set name>**
  The character set for the commands specified in **-cmdfile**.
  The character set name must be a supported charset as defined in java.nio.charset.Charset. For example, US-ASCII.

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

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

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

```
info system
info proc
```

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

```
jdmpview -outfile out.txt [-overwrite|-append] -cmdfile commands.txt -core <path to core file>
```
When the output file exists, you need to specify the -overwrite option. If you do not, then an error message is shown.

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

```
DTFJView version 4.27.56, using DTFJ version 1.10.27022
Loading image from DTFJ...

For a list of commands, type "help"; for how to use "help", type "help help"
Available contexts (* = currently selected context):

Source : file:/home/test/core.20120228.113859.14043.0001.dmp
   +0 : PID: 14073 : JRE 1.7.0 Linux ppc64-64 build 20120216_102976 (pxp6470_27-20120221_01)

> info system
Machine OS: Linux
Machine name: madras
Machine IP address(es): 9.20.88.155
System memory: 8269398016
Java version :
JRE 1.7.0 Linux ppc64-64 build 20120216_102976 (pxp6470_27-20120221_01 )

> info proc

Native thread IDs:
  14044 14073

Command line arguments
sdk/jre/bin/java -Xdump:system:events=vmstop -version

JIT was enabled for this runtime
AOT enabled, FSD enabled, HCR disabled, JIT enabled

Environment variables:
LESSKEY=/etc/lesskey.bin
LESS_ADVANCED_PREPROCESSOR=no
HOSTTYPE=ppc64
CSEDIT=emacs
G_BROKEN_FILENAMES=1
LESSOPEN=lessopen.sh %s
MINICOM=-c on
PATH=/home/test/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/usr/lib/mit/bin:/usr/lib/mit/sbin
INFODIR=/usr/local/info:/usr/share/info:/usr/info

Commands available in jdmpview:

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

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

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

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

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

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

```
find <pattern>,<start_address>,<end_address>,<memory_boundary>,
     <bytes_to_print>,<matches_to_display>
```

This command searches for `<pattern>` in the memory segment from `<start_address>` to `<end_address>` (both inclusive), and shows the number of matching addresses you specify with `<matches_to_display>`. You can also display the next `<bytes_to_print>` bytes for the last match.

By default, the `find` command searches for the pattern at every byte in the range. If you know the pattern is aligned to a particular byte boundary, you can specify `<memory_boundary>` to search every `<memory_boundary>` bytes. For example, if you specify a `<memory_boundary>` of "4", the command searches for the pattern every 4 bytes.

```
findnext
```

Finds the next instance of the last string passed to `find` or `findptr`. It repeats the previous `find` or `findptr` command, depending on which one was issued last, starting from the last match.

```
findptr <pattern>,<start_address>,<end_address>,<memory_boundary>,
      <bytes_to_print>,<matches_to_display>
```

Searches memory for the given pointer. `findptr` searches for `<pattern>` as a pointer in the memory segment from `<start_address>` to `<end_address>` (both inclusive), and shows the number of matching addresses you specify with `<matches_to_display>`. You can also display the next `<bytes_to_print>` bytes for the last match.

By default, the `findptr` command searches for the pattern at every byte in the range. If you know the pattern is aligned to a particular byte boundary, you can specify `<memory_boundary>` to search every `<memory_boundary>` bytes. For example, if you specify a `<memory_boundary>` of "4", the command searches for the pattern every 4 bytes.

```
help [command_name]
```

Shows information for a specific command. If you supply no parameters, `help` shows the complete list of supported commands.

```
info thread [*|all|<native_thread_ID>]
```

Displays information about Java and native threads. The following information is displayed for all threads ("*"), or the specified thread:

- Thread id
- Registers
- Stack sections
- Thread frames: procedure name and base pointer
- Thread properties: list of native thread properties and their values. For example: thread priority.
- Associated Java thread, if applicable:
  - Name of Java thread
  - Address of associated java.lang.Thread object
  - State (shown in JVMTI and java.lang.Thread.State formats)
  - The monitor the thread is waiting for
  - Thread frames: base pointer, method, and filename:line
If you supply no parameters, the command shows information about the current thread.

**info system**
Displays the following information about the system that produced the core dump:
- amount of memory
- operating system
- virtual machine or virtual machines present

**info class [<class_name>] [-sort:<name>|<count>|<size>]**
Displays the inheritance chain and other data for a given class. If a class name is passed to **info class**, the following information is shown about that class:
- name
- ID
- superclass ID
- class loader ID
- modifiers
- if the class is a packed class
- 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.
- if the class is a packed class
- the class name
- the total number of instances of all classes.
- the total size of all objects.

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

**info proc**
Displays threads, command-line arguments, environment variables, and shared modules of the current process.

**Note:** To view the shared modules used by a process, use the **info sym** command.

**info jitm**
Displays JIT compiled methods and their addresses:
- Method name and signature
- Method start address
- Method end address

**info lock**
Displays a list of available monitors and locked objects

**info sym**
Displays a list of available modules. For each process in the address spaces, this command shows a list of module sections for each module, their start and end addresses, names, and sizes.

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

**info heap [x|<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: Default019fce8
0x8e76a8 is within heap segment: 8b0000 -- cb0000
0x8e76a8 is start of an object of type java/lang/Thread
```

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

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Unit</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>Byte</td>
<td>8-bit</td>
</tr>
<tr>
<td>h</td>
<td>Half word</td>
<td>16-bit</td>
</tr>
<tr>
<td>w</td>
<td>Word</td>
<td>32-bit</td>
</tr>
<tr>
<td>g</td>
<td>Giant word</td>
<td>64-bit</td>
</tr>
</tbody>
</table>

\(x/J \ [\texttt{\langle class\_name\rangle} | \texttt{\langle 0xaddr\rangle} | \texttt{\langle class\_name\rangle packed @ \langle 0xaddr\rangle}]\)

Displays information about a particular object, or all objects of a class. If \texttt{\langle class\_name\rangle} 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. Use the \texttt{\langle class\_name\rangle packed @ \langle 0xaddr\rangle} parameter to specify a nested packed object. For more information about using this parameter, see \texttt{``Support for packed objects''} on page \texttt{\textit{377}}.

\textbf{Note:} This command ignores the number of items and unit size passed to it by the \(x/\) command.

\(x/D \ \langle 0xaddr\rangle\)

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.

\textbf{Note:} This command uses the number of items and unit size passed to it by the \(x/\) command.

\(x/X \ \langle 0xaddr\rangle\)

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.

\textbf{Note:} This command uses the number of items and unit size passed to it by the \(x/\) command.

\(x/K \ \langle 0xaddr\rangle\)

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 \texttt{info thread} command.

\textbf{Note:} This command uses the number of items and unit size passed to it by the \(x/\) command.
Example session:

This example session illustrates a selection of the commands available and their use.

In the example session, some lines have been removed for clarity (and terseness).

User input is prefaced by a greater than symbol (>).

bash-3.2# jps | grep -c "core.jar" 20130713-000001-0001.dmp
For a list of commands, type "help", for how to use "help", type "help help"
Available contexts (+ = currently selected context)
Source: /CF/JavaCore/20130713-000001-0001.dmp
> help
displays the next section of memory in heapdump-like format
  - displays the previous section of memory in heapdump-like format
cd changes the current working directory, used for log files
class [context id] switch to the selected context
dealock displays information about deallocks if there are any
exit Exit the application
files displays memory for a given string
findnext finds the next instance of the last string passed to "find"
findprev searches memory for the given printer
heapdump generates a PDF or classic format heapdump
  [command name] displays list of commands or help for a specific command
heapdump -<hexdump> outputs a section of memory in a hexdump-like format
info <component> information about the specified component
info class <Java class name> provides information about the specified Java class
info heap [-heap name] displays information about heap heaps
info jvm Displays JVM methods and their addresses
info lock outputs a list of system monitors and locked objects
info map outputs a list of all memory segments in the address space
info mod Displays module information
info proc Displays short form of info process
info process Displays thread, command line arguments, environment
info sym Displays short form of info system
info system Displays information about the system_heap dump file
info thread Displays threads, command line arguments, environment
log [name level] display control instances of Java.util.logging.Logger
open [path to core or zip] opens the specified core or zip file
plugins Displays plugins management commands
  list Shows the list of loaded plugins for the current context
  reload Reload plugins for the current context
  showpath Shows the DTrace plugin search path for the current context
  walkpath Set the DTrace view plugin search path for the current context
pmd displays the current processing directory
quit Exit the application
set [logging] [heapdump] Sets options for the specified command
set heapdump Configures heapdump format, file name and multiple heap support
set logging Configures若干 heapdump-related parameters, starts/stops logging
  on turn on logging
  off turn off logging
  file turn on logging
swpump Controls the swpumping of log files
show [logging] heapdump Displays the current log set options for a command
show heapdump Displays heapdump settings
show logging Shows the current logging options
  what displays information about what is stored at the given memory address
  @<hex address> displays the integer at the specified address
  @<object address> <class name> displays information about a particular object or all objects of a class
  @<hex address> displays the specified memory section as if it were a stack frame parameters
  @<hex address> displays the hex value of the bytes at the specified address

> set logging file log.txt
logging turned on; outputting to /home/test/log.txt

> info system
Machine OS: AIX
Hypervisor: PowerVM
Machine name: jalfrezi.hursley.ibm.com
Machine IP address(es): 9.20.189.136
PowerVM jalfrezi.hursley.ibm.com
bash-3.2$
Chapter 10. Troubleshooting and support

391

thread id: 11796835

registers:

<table>
<thead>
<tr>
<th>register</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cr0</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>cr2</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>cr3</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>cr4</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>cr5</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>cr6</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>cr7</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>gpr0</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>gpr1</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>gpr2</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>gpr3</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>fpscr</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>gpr15</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>gpr16</td>
<td>0x0000000000000000</td>
</tr>
</tbody>
</table>

properties:
current/last signal taken: 0
scheduling policy: Se
current effective priority: 0
suspended count: 1
thread state: 2
type of thread unit: 0
processor usage: 0.0000%
thread flags: 400000
associatedJavaThread:
name: DestophassAW helper thread
Thread object: java/lang/Thread @ 0x57339b08
Priority: 5
Thread State: RUNNABLE
JVM state: ALIVE RUNNABLE
Java stack frames: no frames to print

Java bytecode:

Bytecode range(s): [0x0000000000000000, 0x0000000000000000]
jdmpview commands quick reference:

A short list of the commands you use with jdmpview.

The following table shows the jdmpview - quick reference:

<table>
<thead>
<tr>
<th>Command</th>
<th>Sub-command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td></td>
<td>Displays a list of commands or help for a specific command.</td>
</tr>
<tr>
<td>info</td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread</td>
<td></td>
<td>Displays information about Java and native threads.</td>
</tr>
<tr>
<td>system</td>
<td></td>
<td>Displays information about the system the core dump is from.</td>
</tr>
<tr>
<td>class</td>
<td></td>
<td>Displays the inheritance chain and other data for a given class.</td>
</tr>
<tr>
<td>proc</td>
<td></td>
<td>Displays threads, command line arguments, environment variables, and shared modules of current process.</td>
</tr>
<tr>
<td>jitm</td>
<td></td>
<td>Displays JIT compiled methods and their addresses.</td>
</tr>
<tr>
<td>lock</td>
<td></td>
<td>Displays a list of available monitors and locked objects.</td>
</tr>
<tr>
<td>sym</td>
<td></td>
<td>Displays a list of available modules.</td>
</tr>
<tr>
<td>mmap</td>
<td></td>
<td>Displays a list of all memory segments in the address space.</td>
</tr>
<tr>
<td>heap</td>
<td></td>
<td>Displays information about all heaps or the specified heap.</td>
</tr>
<tr>
<td>heapdump</td>
<td></td>
<td>Generates a Heapdump to a file. You can select which Java heaps should be dumped by listing the heap names, separated by spaces.</td>
</tr>
<tr>
<td>hexdump</td>
<td></td>
<td>Displays a section of memory in a hexdump-like format.</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td>Displays the next section of memory in hexdump-like format.</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>Displays the previous section of memory in hexdump-like format.</td>
</tr>
<tr>
<td>whatis</td>
<td></td>
<td>Displays information about what is stored at the given memory address.</td>
</tr>
<tr>
<td>find</td>
<td></td>
<td>Searches memory for a given string.</td>
</tr>
<tr>
<td>findnext</td>
<td></td>
<td>Finds the next instance of the last string passed to “find”.</td>
</tr>
<tr>
<td>findptr</td>
<td></td>
<td>Searches memory for the given pointer.</td>
</tr>
<tr>
<td>x/</td>
<td>(examine)</td>
<td><strong>Examine</strong> works like x/ in gdb, including the use of defaults: passes the number of items to display and the unit size (b for byte (8-bit), h for half word (16-bit), w for word (32-bit), g for giant word (64-bit)) to the sub-command. For example x/12bd.</td>
</tr>
<tr>
<td>J</td>
<td></td>
<td>Displays information about a particular object or all objects of a class.</td>
</tr>
<tr>
<td>Command</td>
<td>Sub-command</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>D</td>
<td>Displays the integer at the specified address.</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>Displays the hex value of the bytes at the specified address.</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>Displays the specified memory section as if it were a stack frame.</td>
<td></td>
</tr>
<tr>
<td>deadlock</td>
<td>Displays information about deadlocks if there are any set.</td>
<td></td>
</tr>
<tr>
<td>set heapdump</td>
<td>Configures Heapdump generation settings.</td>
<td></td>
</tr>
<tr>
<td>set logging</td>
<td>Configures logging settings, starts logging, or stops logging. This allows the results of commands to be logged to a file.</td>
<td></td>
</tr>
<tr>
<td>show heapdump</td>
<td>Displays the current values of heapdump settings.</td>
<td></td>
</tr>
<tr>
<td>show logging</td>
<td>Displays the current values of logging settings.</td>
<td></td>
</tr>
<tr>
<td>cd</td>
<td>Changes the current working directory, used for log files.</td>
<td></td>
</tr>
<tr>
<td>pwd</td>
<td>Displays the current working directory.</td>
<td></td>
</tr>
<tr>
<td>quit</td>
<td>Exits the core file viewing tool; any log files that are currently open are closed before the tool exits.</td>
<td></td>
</tr>
</tbody>
</table>

**Tracing Java applications and the JVM**

JVM trace is a trace facility that is provided in all IBM-supplied JVMs with minimal affect on performance. Trace data can be output in human-readable or in compressed binary formats. The JVM provides a tool to process and convert the compressed binary data and into a readable format.

JVM trace data might contain application data, including the contents of Java objects. If you require a dump that does not contain this application data, see "Using Javadump" on page 340 or "Using Heapdump" on page 363.

Tracing is enabled by default, together with a small set of trace points going to memory buffers. You can enable tracepoints at run time by using levels, components, group names, or individual tracepoint identifiers.

This chapter describes JVM trace in:

- "What can be traced?" on page 394
- "Types of tracepoint" on page 394
- "Default tracing" on page 395
- "Where does the data go?" on page 396
- "Controlling the trace" on page 398
- "Running the trace formatter" on page 417
- "Determining the tracepoint ID of a tracepoint" on page 419
- "Application trace" on page 420
- "Using method trace" on page 423

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: Snapnnnn.yyyymmdd.hhmmss.th.proces.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 399 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}
```

When startup is complete, the equivalent of the following command line is added to enable level 2 trace points:

```
-Xtrace:maximal=all{level2}
```

Level 2 is used for default tracing that would produce too much data during the startup of the JVM. If you set other trace options on the command line, or before the JVM finishes startup (through use of JVMTI or the `com.ibm.jvm.Trace` API), the level 2 trace points are enabled just before your trace options are processed. This behavior ensures that the default level 2 trace points do not override any changes that you specify.

The data generated by the tracepoints is continuously captured in wrapping memory buffers for each thread. (For information about specific options, see “Detailed descriptions of trace options“ on page 399.)

You can find tracepoint information in the following diagnostics data:

- System memory dumps, extracted by using `jdpview`.
- Snap traces, generated when the JVM encounters a problem or an output file is specified. "Using dump agents“ on page 321 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 398 for more details.

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

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

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

Where does the data go?
Trace data can be written to a number of locations.

Trace data can go into:

- Memory buffers that can be dumped or snapped when a problem occurs
- One or more files that are using buffered I/O
- An external agent in real time
- stderr in real time
- Any combination of the other items in this list

Writing trace data to memory buffers:

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

Buffers are allocated on a per-thread principle. This principle removes contention between threads, and prevents trace data for an individual thread from being mixed in with trace data from other threads. For example, if one particular thread is not being dispatched, its trace information is still available when the buffers are
dumped or snapped. Use the `-Xtrace:buffers=<size>` option to control the size of the buffer allocated to each thread. Buffers allocated to a thread are discarded when that thread terminates.

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

To examine the trace data captured in these memory buffers, you must snap or dump the data, then format the buffers.

**Snapping buffers**

Under default conditions, a running JVM collects a small amount of trace data in special wraparound buffers. This data is sent to a snap trace file under certain conditions:

- An uncaught `OutOfMemoryError` occurs.
- An operating system signal or exception occurs.
- The `com.ibm.jvm.Trace.snap()` Java API is called.
- The JVMRI `TraceSnap` function is called.

The resulting snap trace file is placed into the current working directory, with a name in the format `Snapnnnn.yyyyMMdd.hhmsstht.process.trc`, where `nnnn` is a sequence number reset to 0001 at JVM startup, `yyyyMMdd` is the current date, `hhmsstht` is the current time, and `process` is the process identifier. This file is in a binary format, and requires the use of the supplied trace formatter so that you can read it.

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

**Extracting buffers from system dump**

You can extract the buffers from a system dump core file by using the Dump Viewer.

**Writing trace data to a file:**

You can write trace data to a file continuously as an extension to the in-storage trace, but, instead of one buffer per thread, at least two buffers per thread are allocated, and the data is written to the file before wrapping can occur.

This allocation allows the thread to continue to run while a full trace buffer is written to disk. Depending on trace volume, buffer size, and the bandwidth of the output device, multiple buffers might be allocated to a given thread to keep pace with trace data that is being generated.

A thread is never stopped to allow trace buffers to be written. If the rate of trace data generation greatly exceeds the speed of the output device, excessive memory usage might occur and cause out-of-memory conditions. To prevent this, use the `nodynamic` option of the `buffers` trace option. For long-running trace runs, a `wrap` option is available to limit the file to a given size. It is also possible to create a sequence of files when the trace output will move back to the first file once the sequence of files are full. See the `output` option for details. You must use the trace formatter to format trace data from the file.

Because trace data is buffered, if the JVM does not exit normally, residual trace buffers might not be flushed to the file. If the JVM encounters an unrecoverable
error, the buffers can be extracted from a system dump if that is available. When a snap file is created, all available buffers are always written to it.

**External tracing:**

You can route trace to an agent by using the JVMTI RegisterTracePointSubscriber() API.

For more information about this capability, see "Subscribing to JVM tracepoints" on page 503.

This mechanism allows a callback routine to be called immediately when any of the selected tracepoints are found, without buffering the trace results. The trace data is in formatted text form. To route selected tracepoints to the JVMTI listener, you must specify the option `-Xtrace:external=<tracepointer_specification>`. For more information about configuring trace options, see "Options that control tracepoint activation" on page 402.

**Tracing to stderr:**

For reduced volume or non-performance-critical tracing, the trace data can be formatted and routed to stderr immediately without buffering.

For more information, see "Using method trace" on page 423.

**Trace combinations:**

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

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

**Controlling the trace**

You have several ways by which you can control the trace.

You can control the trace in several ways by using:

- The `-Xtrace` options when launching the JVM, including trace trigger events
- A trace properties file
- `com.ibm.jvm.Trace` API
- JVMTI and JVMRI from an external agent

**Note:**

1. The specification of trace options is cumulative. Multiple `-Xtrace` options are accepted on the command line and they are processed in order starting with the option that is closest to the `-Xtrace` string. Each option adds to the previous options (and to the default options), as if they had all been specified in one long comma-separated list in a single option. This cumulative specification is consistent with the related `-Xdump` option processing.

2. Some trace options are enabled by default. For more information, see "Default tracing" on page 395. To disable the defaults, use the `-Xtrace:none` option.
3. Many diagnostic tools start a JVM. When using the IBM_JAVA_OPTIONS environment variable to 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" 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**
properties option:

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.

properties[=<filename>]

If <filename> is not specified, a default name of IBMTRACE.properties is searched for in the current directory.

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. Nesting is not supported; that is, the file cannot contain a properties option. If any error is found when the file is accessed, JVM initialization fails with an explanatory error message and return code. All the options that are in the file are processed in the sequence in which they are stored in the file, before the next option that is obtained through the normal mechanism is processed. Therefore, a command-line property always overrides a property that is in the file.

An existing restriction means that you cannot leave properties that have the form <name>=<value> to default if they are specified in the property file; that is, you must specify a value, for example maximal=all.

Another restriction means that properties files are sensitive to white space. Do not add white space before, after, or within the trace options.
You can make comments as follows:
// This is a comment. Note that it starts in column 1

Examples
- Use IBMTRACE.properties in the current directory:
  -Xtrace:properties
- Use trace.prop in the current directory:
  -Xtrace:properties=trace.prop
- Use c:\trc\gc\trace.props:
  -Xtrace:properties=c:\trc\gc\trace.props

Here is an example property file:
```
minimal=all
// maximal=j9mm
maximal=j9shr
buffers=20k
output=c:\traces\classloader.trc
print=tpnid(j9vm.23-25)
```

buffers option:

You can modify the size of the buffers to change how much diagnostic output is provided in a snap dump. This buffer is allocated for each thread that makes trace entries.

The buffers trace option can be specified in two ways:
- **buffers=dynamic|nodynamic**
- **buffers=nnnk|nnnm,[dynamic|nodynamic]**

If external trace is enabled, the number of buffers is doubled; that is, each thread allocates two or more buffers. The same buffer size is used for state and exception tracing, but, in this case, buffers are allocated globally. The default is 8 KB per thread.

The dynamic and nodynamic options have meaning only when tracing to an output file. If dynamic is specified, buffers are allocated as needed to match the rate of trace data generation to the output media. Conversely, if nodynamic is specified, a maximum of two buffers per thread is allocated. The default is dynamic. The dynamic option is effective only when you are tracing to an output file.

Note: If nodynamic is specified, you might lose trace data if the volume of trace data that is produced exceeds the bandwidth of the trace output file. Message UTE115 is issued when the first trace entry is lost, and message UTE018 is issued at JVM termination.

Examples
- Dynamic buffering with increased buffer size of 2 MB per thread:
  -Xtrace:buffers=2m

  or in a properties file:
  ```
  buffers=2m
  ```

- Trace buffers limited to two buffers per thread, each of 128 KB:
  -Xtrace:buffers={128k,nodynamic}
or in a properties file:
buffers=128k,nodynamic

- Trace using default buffer size of 8 KB, limited to two buffers per thread:
  -Xtrace:buffers=nodynamic

  or in a properties file:
  buffers=nodynamic

Options that control tracepoint activation:

These options control which individual tracepoints are activated at run time and the implicit destination of the trace data.

In some cases, you must use them with other options. For example, if you specify maximal or minimal tracepoints, the trace data is put into memory buffers. If you are going to send the data to a file, you must use an output option to specify the destination file name.

### Command-Line Options

- `minimal` = ![tracepoint_specification][,...]
- `maximal` = ![tracepoint_specification][,...]
- `count` = ![tracepoint_specification][,...]
- `print` = ![tracepoint_specification][,...]
- `iprint` = ![tracepoint_specification][,...]
- `exception` = ![tracepoint_specification][,...]
- `external` = ![tracepoint_specification][,...]
- `none` = ![tracepoint_specification][,...]

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 you use 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 time stamp 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. These types of trace are 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. When the JVM ends, all nonzero 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 that are used in the default trace configuration, use the following command:

```
-Xtrace:count=all[level1],count=j9mm(gclogger)
```

**print**

The `print` option causes the specified tracepoints to be routed to stderr in real time. The JVM tracepoints are formatted by using J9TraceFormat.dat. The class library tracepoints are formatted by TraceFormat.dat. J9TraceFormat.dat and TraceFormat.dat are in sdk/jre/lib directory and are automatically found by the Runtime environment.

**iprint**

The `iprint` option is the same as the `print` option, but uses indenting to format the trace.

**exception**

When `exception` trace is enabled, the trace data is collected in internal buffers that are separate from the normal buffers. These internal buffers can then be written to a snap file or written to the file that is specified in an `exception.output` option.

The `exception` option allows low-volume tracing in buffers and files that are distinct from the higher-volume information that `minimal` and `maximal` tracing have provided. In most cases, this information is exception-type data, but you can use this option to capture any trace data that you want.

This form of tracing is channeled through a single set of buffers, as opposed to the buffer-per-thread approach for normal trace. 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 404](#).

**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 395](#). You can send additional tracepoints to the exception buffers or turn off the garbage collection tracepoints. Changing the exception trace buffers alters 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, which 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 routes trace data to trace listeners, which are registered by using the JVMTI RegisterTracePointSubscriber() and DeregisterTracePointSubscriber() APIs.

**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 still loads.
If you specify other tracepoints without specifying `-Xtrace:none`, the tracepoints are added to the default set.

**Examples**

- To apply default options, specify the following command:
  ```
  java
  ```
- This option has no effect apart from ensuring that the trace engine is loaded, which is the default behavior:
  ```
  java -Xtrace
  ```
- When using this option, the trace engine is not loaded:
  ```
  java -Xtrace:none
  ```
- When using this option the trace engine is loaded, but no tracepoints are captured:
  ```
  java -Xtrace:none=all
  ```
- When using this option, defaults are applied with the addition of printing for `j9vm.209`:
  ```
  java -Xtrace:iprint=j9vm.209
  ```
- When using this option, defaults are 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}
  ```
- To print only for `j9vm.209`, use only this command syntax:
  ```
  java -Xtrace:none -Xtrace:iprint=j9vm.209
  ```
- To print only for `j9vm.209`, use only this command syntax:
  ```
  java -Xtrace:none,iprint=j9vm.209
  ```
- To turn on default tracing for all components except `j9vm`, with printing for `j9vm.209`, use this command syntax:
  ```
  java -Xtrace:none=j9vm,iprint=j9vm.209
  ```
- To turn on default tracing for all components except `j9vm`, with printing for `j9vm.209`, use this command syntax:
  ```
  java -Xtrace:none=j9vm -Xtrace:iprint=j9vm.209
  ```
- With this option, there is no tracing for the component `j9vm`. The option `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:

```
[1]<component>[[<group>]] or [1]<component>[[<type>]] or [1]<tracepoint_id>[,<tracepoint_id>]
```

where:

- `!` is a logical not. That is, the tracepoints that are in a specification starting with `!` are turned off.

- `<component>` is a Java component, as detailed in [Table 19 on page 405](#). To include all Java components, specify `all`.  

---

Table 19. Java components

<table>
<thead>
<tr>
<th>Component name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>avl</td>
<td>VM AVL tree support</td>
</tr>
<tr>
<td>io</td>
<td>Class library java.io native code</td>
</tr>
<tr>
<td>j9bcu</td>
<td>VM byte code utilities</td>
</tr>
<tr>
<td>j9bcverify</td>
<td>VM byte code verification</td>
</tr>
<tr>
<td>j9codertvm</td>
<td>VM byte code run time</td>
</tr>
<tr>
<td>j9dmp</td>
<td>VM dump</td>
</tr>
<tr>
<td>j9jcl</td>
<td>VM class libraries</td>
</tr>
<tr>
<td>j9jit</td>
<td>VM JIT interface</td>
</tr>
<tr>
<td>j9jni</td>
<td>VM JNI support</td>
</tr>
<tr>
<td>j9jvmti</td>
<td>VM JVMTI support</td>
</tr>
<tr>
<td>j9mm</td>
<td>VM memory management</td>
</tr>
<tr>
<td>j9prt</td>
<td>VM port library</td>
</tr>
<tr>
<td>j9scar</td>
<td>VM class library interface</td>
</tr>
<tr>
<td>j9shr</td>
<td>VM shared classes</td>
</tr>
<tr>
<td>j9trace</td>
<td>VM trace</td>
</tr>
<tr>
<td>j9util</td>
<td>VM utilities</td>
</tr>
<tr>
<td>j9vm</td>
<td>VM general</td>
</tr>
<tr>
<td>j9vmutil</td>
<td>VM utilities</td>
</tr>
<tr>
<td>j9vrb</td>
<td>VM verbose stack walker</td>
</tr>
<tr>
<td>JVERBS</td>
<td>Class library jVerbs native code</td>
</tr>
<tr>
<td>map</td>
<td>VM mapped memory support</td>
</tr>
<tr>
<td>mt</td>
<td>Java methods (see note)</td>
</tr>
<tr>
<td>net</td>
<td>Class library TCP/IP networking native code</td>
</tr>
<tr>
<td>pool</td>
<td>VM storage pool support</td>
</tr>
<tr>
<td>rpc</td>
<td>VM RPC support</td>
</tr>
<tr>
<td>simplepool</td>
<td>VM storage pool support</td>
</tr>
<tr>
<td>sunvmi</td>
<td>VM class library interface</td>
</tr>
</tbody>
</table>

Note: When specifying the mt component you must also specify the methods option.

<group> is a tracepoint group. A group is a set of tracepoints that are defined within a component, therefore each group is associated with one or more components as follows:

Table 20. Tracepoint groups and associated components

<table>
<thead>
<tr>
<th>Component name or names</th>
<th>Group name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>j9mm</td>
<td>gclogger</td>
<td>A set of tracepoints that record each garbage collection cycle. Equivalent to -verbose:gc output</td>
</tr>
</tbody>
</table>
Table 20. Tracepoint groups and associated components (continued)

<table>
<thead>
<tr>
<th>Component name or names</th>
<th>Group name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>j9prt</td>
<td>nlsmessage</td>
<td>A set of tracepoints that record each NLS message that is issued by the JVM</td>
</tr>
<tr>
<td>j9jcl j9vm</td>
<td>verboseclass</td>
<td>A set of tracepoints that record each class as it is loaded. Equivalent to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-verbose:class output</td>
</tr>
<tr>
<td>j9jni j9vm</td>
<td>checkjni</td>
<td>A set of tracepoints that record JNI function checks. Equivalent to -Xcheck:jni</td>
</tr>
<tr>
<td></td>
<td></td>
<td>output</td>
</tr>
<tr>
<td>j9vm</td>
<td>checkmemory</td>
<td>A set of tracepoints that record memory checks. Equivalent to -Xcheck:memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>output</td>
</tr>
<tr>
<td>j9vm</td>
<td>checkvm</td>
<td>A set of tracepoints that record VM checks. Equivalent to -Xcheck:vm output</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j9jit</td>
<td>verbose</td>
<td>A set of tracepoints that record JIT compiler configuration and method</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compilation. Equivalent to -Xjit:verbose output</td>
</tr>
<tr>
<td>mt</td>
<td>compiledMethods</td>
<td>A set of tracepoints that record compiled Java methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mt</td>
<td>nativeMethods</td>
<td>A set of tracepoints that record Java native methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mt</td>
<td>staticMethods</td>
<td>A set of tracepoints that record Java static methods</td>
</tr>
</tbody>
</table>

**<type>** is the tracepoint type. The following types are supported:
- Entry
- Exit
- Event
- Exception
- Mem

**<tracepoint_id>** is the tracepoint identifier. The tracepoint identifier constitutes the component name of the tracepoint, followed by its integer number inside that component. For example, j9mm.49, j9shr.20-29, j9vm.15. To understand these numbers, see “Determining the tracepoint ID of a tracepoint” on page 419.

Some tracepoints can be both an exit and an exception; that is, the function ended with an error. If you specify either exit or exception, these tracepoints are included.

The following tracepoint specification used in Java 5.0 and earlier IBM SDKs is still supported:

\[ ![tpnid\{<tracepoint_id>[,...]\}] \]
Examples

- All tracepoints:
  -Xtrace:maximal=all

- All tracepoints except j9vrb and j9trc:
  -Xtrace:minimal={all},minimal=\{!j9vrb,j9trc\}

- All entry and exit tracepoints in j9bcu:
  -Xtrace:maximal=\{j9bcu\{entry\},j9bcu\{exit\}\}

- All tracepoints in j9mm except tracepoints 20-30:
  -Xtrace:maximal=j9mm,maximal=\{!j9mm.20-30\}

- Tracepoints j9prt.5 through j9prt.15:
  -Xtrace:print=j9prt.5-15

- All j9trc tracepoints:
  -Xtrace:count=j9trc

- All entry and exit tracepoints:
  -Xtrace:external={all\{entry\},all\{exit\}}

Trace levels:

Tracepoints have been assigned levels 0 through 9 that are based on the importance of the tracepoint.

A level 0 tracepoint is the most important. It is reserved for extraordinary events and errors. A level 9 tracepoint is in-depth component detail. To specify a given level of tracing, the level0 through level9 keywords are used. You can abbreviate these keywords to l0 through l9. For example, if level5 is selected, all tracepoints that have levels 0 through 5 are included. Level specifications do not apply to explicit tracepoint specifications that use the TPNID keyword.

The level is provided as a modifier to a component specification, for example:

-Xtrace:maximal={all\{level5\}}

or

-Xtrace:maximal=\{j9mm\{L2\},j9trc,j9bcu\{level19\},all\{level11\}\}

In the first example, tracepoints that have a level of 5 or less are enabled for all components. In the second example, all level 1 tracepoints are enabled. All level2 tracepoints in j9mm are enabled. All tracepoints up to level 9 are enabled in j9bcu.

Note: The level applies only to the current component. If multiple trace selection components are found in a trace properties file, the level is reset to the default for each new component.

Level specifications do not apply to explicit tracepoint specifications that use the TPNID keyword.

When the not operator is specified, the level is inverted; that is, \(!j9mm\{level15\}\) disables all tracepoints of level 6 or greater for the j9mm component. For example:

-Xtrace:print=\{all\},print=\{!j9trc\{15\},j9mm\{16\}\}

enables trace for all components at level 9 (the default), but disables level 6 and higher for the locking component, and level 7 and higher for the storage component.
Examples

- Count the level zero and level one tracepoints matched:
  -Xtrace:count=all{L1}

- Produce maximal trace of all components at level 5 and j9mm at level 9:
  -Xtrace:maximal={all{level5},j9mm{L9}}

- Trace all components at level 6, but do not trace j9vrb at all, and do not trace the entry and exit tracepoints in the j9trc component:
  -Xtrace:minimal={all{l6}},minimal={!j9vrb,j9trc{entry},j9trc{exit}}

methods option:

Using method trace provides a complete and potentially large diagnosis of code paths inside your application and the system classes. Use wildcards and filtering to control method trace so that you can focus on the sections of code that interest you.

methods=<method_specification>[,<method_specification>]

Specify one or more method specifications.

Method trace can trace method entry and method exit.

The methods parameter is defined as:

\[
\text{methods} = \{ [\text{[!]} \text{[package]}/\text{class}[\text{[()]}]}\text{[method]}\text{[\text{[()]}]}\}
\]

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 option" on page 410.
Examples

- **Tracing entry and exit of all methods in a given class:**
  
  ```java
  -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:**
  
  ```java
  -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:**
  
  ```java
  -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:**
  
  ```java
  -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**
  
  ```java
  -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**
  
  ```java
  -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**
  
  ```java
  -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
```
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 option:**

Use the output option to send trace data to *<filename>*. You can optionally limit the size of the file, with multiple files being created as the threshold is reached.

`output=<filename>[,size[,generations]]`

If *<filename>* 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 sequentially as 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 option:**

Use the `exception.output` option to redirect exception trace data to <filename>.  

`exception.output=<filename>[, nnnm]`

If the file does not already exist, it is created automatically. Otherwise, it is overwritten.

Optionally, you can limit the file to nnn MB. When the limit is reached, the output 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 process ID number 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 option:

The resume option resumes tracing globally.

The suspend and resume options 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 option:

This trace option determines whether tracing is enabled for each thread.

resumecount=<count>

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 option is used with the trigger option. For more information, see “trigger option” on page 414.

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 option:**

Specify how long the sleep lasts when using the sleep trigger action.

```
sleeptime=<n> | <a> | <b>
```

Where:

- `n` Sleep for `n` milliseconds.
- `a` Sleep for `a` milliseconds.
- `b` Sleep for `b` seconds.

The default length of time is 30 seconds. If no units are specified, the default time unit is milliseconds.

**stackdepth option:**

Use this option to limit the maximum number of stack frames reported by the jstacktrace trace trigger action.

```
stackdepth=<n>
```

Where:

- `n` Record `n` stack frames. All stack frames are recorded by default.

**suspend option:**

Suspends tracing globally for all threads and all forms of tracing but leaves tracepoints activated.

**Example**

- To suspend tracing, use the following syntax:
  ```
  -Xtrace:suspend
  ```

**suspendcount option:**

This trace option determines whether tracing is enabled for each thread.

```
suspendcount=<count>
```

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 option" on page 414.

**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 option:**

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.

**trigger=<clause>[,<clause>]**

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.

`<methodspec>` is the specification of a Java method, consisting of a class and a method name separated by a dot. For example, specify `com/HelloWorld.main`. If the class is in a package, the package name must be included, separated by slashes. For example, specify `java/lang/String.getBytes`. A wildcard `*` can be used at the start or end of the class and method names, or both. For example, you can specify `*/String.get*`. To specify a constructor method, use `<init>` as the method name. Method signatures cannot be specified, so a method specification applies to all overloaded methods.

**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.

**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.

**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 363.
javadump
Produce a Javadump. See “Using Javadump” on page 340.

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 option” on page 413.

resume
Resume all tracing (except for threads that are suspended by the action of the resumecount property and Trace.suspendThis() calls).

resumethis
Decrement the suspend count for this thread. If the suspend count is zero or less, resume tracing for this thread.

segv
Cause a segmentation violation. (Intended for use in debugging.)

sleep
Delay the current thread for a length of time controlled by the sleeptime option. The default is 30 seconds. See “sleeptime option” on page 413.

snap
Snap all active trace buffers to a file in the current working directory. The file name has the format: Snapnnnn.yyyymmdd.hhmssth.ppppp.trc, where nnnn is the sequence number of the snap file since JVM startup, yyyymmdd is the date, hhmssth 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 373.

Examples
• To produce a Java dump when a method is entered:
  -Xtrace:trigger=method{java/lang/String.getBytes,javadump}
• To produce a system dump when a method is entered:
  -Xtrace:trigger=method{java/lang/String.getBytes,sysdump}
• To produce a Java dump when a class constructor is called:
  "-Xtrace:trigger=method{java/lang/Thread.<init>,javadump}"

This trace option is enclosed in quotation marks to avoid unwanted shell expansion of some of the characters.
• To produce a Java dump when a class static initializer is called:
  "-Xtrace:trigger=method{java/lang/Thread.<clinit>,javadump}"

This trace option is enclosed in quotation marks to avoid unwanted shell expansion of some of the characters.
• To produce a Java dump when a method is entered 1000 times and 1001 times:
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 trace (all threads) while the application is active; that is, not starting or shutting down. (The application name is "HelloWorld"):

-Xtrace:suspend,trigger=method\{HelloWorld.main,resume,suspend\}

To print a Java stack trace to the console when the mycomponent.1 tracepoint is reached:

-Xtrace:print=mycomponent.1,trigger=tpnid\{mycomponent.1,jstacktrace\}

To write a Java stack trace to the trace output file when the Sample.code() method is called:

-Xtrace:maximal=mt,output=trc.out,methods={mycompany/mypackage/Sample.code},trigger=method\{mycompany/mypackage/Sample.code,jstacktrace\}

what option:

Use this option to show the current trace settings.

Example

-Xtrace:what

Example output:

Trace engine configuration
-----------------------------
-Xtrace:FORMAT=C:\Java\jre\bin;C:\Java\jre\lib;.
-Xtrace:LIBPATH=C:\Java\jre\bin
-Xtrace:MAXIMAL=all\{level1\}
-Xtrace:EXCEPTION=j9mm\{gclogger\}
-Xtrace:what
-----------------------------

Using the Java API:

You can dynamically control trace in a number of ways from a Java application by using the com.ibm.jvm.Trace class.

Activating and deactivating tracepoints

int set(String cmd);

The Trace.set() method allows a Java application to select tracepoints dynamically. For example:

Trace.set("iprint=all");

The syntax is the same as that used in a trace properties file for the print, iprint, count, maximal, minimal and external trace options.

A single trace command is parsed per invocation of Trace.set, so to achieve the equivalent of

-Xtrace:maximal=j9mm,iprint=j9shr
two calls to Trace.set are needed with the parameters maximal=j9mm and iprint=j9shr

Obtaining snapshots of trace buffers
void snap();

You must have activated trace previously with the **maximal** or **minimal** options and without the **out** option.

**Suspending or resuming trace**

void suspend();

The Trace.suspend() method suspends tracing for all the threads in the JVM.

void resume();

The Trace.resume() method resumes tracing for all threads in the JVM. It is not recursive.

void suspendThis();

The Trace.suspendThis() method decrements the suspend and resume count for the current thread and suspends tracing the thread if the result is negative.

void resumeThis();

The Trace.resumeThis() method increments the suspend and resume count for the current thread and resumes tracing the thread if the result is not negative.

**Running 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 **TraceFormat.dat** and **J9TraceFormat.dat** files, which contain the formatting templates. The formatter produces a file that contains 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 time stamp, thread ID, trace point ID, and trace point data.

To use the trace formatter on a binary trace file type:

```java
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 that is 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 by 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 following files are used: $JAVA_HOME/lib/J9TraceFormat.dat and $JAVA_HOME/lib/TraceFormat.dat

- **-format_time=yes|no**

  Specifies whether to format the time stamps into human readable form. The default is yes.
-help
Displays usage information.

-indent
Indents trace messages at each Entry trace point and outdents trace messages at each Exit trace point. The default is not to indent the messages.

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

-threads=<thread id>[,<thread id>]
Filters the output for the given thread IDs only. thread id is the ID of the thread, which can be specified in decimal or hex (0x) format. Any number of thread IDs can be specified, separated by commas.

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

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

The following example shows output from running the trace formatter command:
C:\test>java com.ibm.jvm.TraceFormat sample.trc
Writing formatted trace output to file sample.trc.fmt
Processing 0.4921875Mb of binary trace data
Completed processing of 6983 tracepoints with 0 warnings and 0 errors

The formatted trace output looks similar to the following extract, which is truncated to show the key areas of information:

Trace Summary

Service level:
JRE 1.7.0 Windows 7 amd64-64 build {pwa6470sr9-20150624_06(SR9)}

JVM startup options:
-xoptionsfile=c:\build\pwa6470sr9-20150624\sdk\jre\bin\compressedrefs\options.default

Processor information:
Arch family: AMD64
Processor Sub-type: Opteron
Num Processors: 8
Word size: 64

Trace activation information:
FORMAT=c:\build\pwa6470sr9-20150624\sdk\jre\lib;
MAXIMAL=all{level1}
EXCEPTION=j9mm{gclogger}
MAXIMAL=all{level2}
output=sample

Trace file header:
JVM start time: 08:58:35.527000000
Generations: 1
Pointer size: 8

Active threads

0x0000000000f155f00 Attach API wait loop
0x0000000000f1b200 Thread-1
0x0000000000f19200 Thread-3

Trace Formatted Data

Time (UTC) Thread ID Tracepoint ID Type Tracepoint Data
08:58:35.5272919 0x0000000000f10500 0x0000000000f10500 0x0000000000f10500 0x0000000000f10500
08:58:35.5273498 0x0000000000f10500 0x0000000000f10500 0x0000000000f10500
08:58:35.5273540 0x0000000000f10500 0x0000000000f10500 0x0000000000f10500
08:58:35.5294696 0x0000000000f10500 0x0000000000f10500 0x0000000000f10500
08:58:35.5360936 0x0000000000f10500 0x0000000000f10500 0x0000000000f10500
08:58:35.5361345 0x0000000000f10500 0x0000000000f10500 0x0000000000f10500
08:58:35.5361364 0x0000000000f10500 0x0000000000f10500 0x0000000000f10500
08:58:35.5361364 0x0000000000f10500 0x0000000000f10500 0x0000000000f10500
08:58:35.5362003 0x0000000000f10500 0x0000000000f10500 0x0000000000f10500
08:58:35.5362003 0x0000000000f10500 0x0000000000f10500 0x0000000000f10500

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 Java runtime environment, and the trace formatter uses these files to convert compressed trace points into readable form.

JVM developers and Service can use the two .dat files to enable formulation of trace point ids and ranges for use under -Xtrace when tracking down problems. The next sample is taken from the beginning of J9TraceFormat.dat, which illustrates how this mechanism works:

```
5.1
j9bcu.0 0 1 1 N Trc_BCU_VMInitStages_Event1 " Trace engine initialized for module j9dyn"
j9bcu.1 2 1 1 N Trc_BCU_internalDefineClass_Entry " >internalDefineClass %p"
j9bcu.2 4 1 1 N Trc_BCU_internalDefineClass_Exit " <internalDefineClass %p ->"
j9bcu.3 2 1 1 N Trc_BCU_createRomClassEndian_Entry " >createRomClassEndian searchFilename=%s"
```

The first line of the .dat file is an internal version number. Following the version number is a line for each tracepoint. Trace point j9bcu.0 maps to Trc_BCU_VMInitStages_Event1 for example and j9bcu.2 maps to Trc_BCU_internalDefineClass_Exit.

The format of each tracepoint entry is:
```
<component.id> <t> <o> <l> <e> <symbol> <template>
```

where:
```
<component.id>
  is the SDK component name.
<t>
  is the tracepoint type (0 through 12), where these types are used:
  • 0 = event
  • 1 = exception
  • 2 = function entry
  • 4 = function exit
  • 5 = function exit with exception
  • 8 = internal
  • 12 = assert

&o>
  is the overhead (0 through 10), which determines whether the tracepoint is compiled into the runtime JVM code.

<l>
  is the level of the tracepoint (0 through 9). High frequency tracepoints, known as hot tracepoints, are assigned higher level numbers.

<e>
  is an internal flag (Y/N) and no longer used.

<symbol>
  is the internal symbolic name of the tracepoint.

<template>
  is a template in double quotation marks that is used to format the entry.
```

For example, if you discover that a problem occurred somewhere close to the issue of Trc_BCU_VMInitStages_Event, you can rerun the application with
```
-Xtrace:print=tpnid{j9bcu.0}. That command will result in an output such as:
```
14:10:42.717+0x41508a00 j9bcu.0 Trace engine initialized for module j9dyn
```

Chapter 10. Troubleshooting and support  419
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:

```java
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
class, the template has a blank character, followed by the format string.

The trace types are defined as static values within the Trace class:

```java
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 l3);
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 %ll.

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 423.

```java
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 422 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
Hello
09:50:29.417 0x2a08a00 084000 > Entering sayHello
09:50:29.427 0x2a08a00 084001 < Exiting sayHello
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 402 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 wildcards 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. Method trace performance has been improved in the J9 VM V2.7 to reduce this. The impact on application throughput is proportional to the number of executed methods that are being traced. A full trace of all methods significantly affects application throughput. Tracing a single method of interest, that is not called continually, should not make a noticeable difference. Additionally, trace output is reasonably large and might require a large amount of drive space. For instance, a full method trace of a “Hello World” application is over 10 MB.

**Running with method trace:**

Control method trace by using the command-line option `-Xtrace:<option>`.
To produce method trace you need to set trace options for the Java classes and methods you want to trace. You also need to route the method trace to the destination you require.

You must set the following two options:

1. Use `-Xtrace:methods` to select which Java classes and methods you want to trace. You can use IBM Monitoring and Diagnostic Tools - Health Center to monitor your application to see which methods should be traced. You can also use the tool to generate the required `-Xtrace` parameters, and view the resulting data.

2. Use either
   - `-Xtrace:print` to route the trace to stderr.
   - `-Xtrace:maximal` and `-Xtrace:output` to route the trace to a binary compressed file using memory buffers.

Use the `methods` parameter to control what is traced. For example, to trace all methods on the String class, set `-Xtrace:methods=java/lang/String.*,print=mt`.

The `methods` parameter is formally defined as follows:

```
-Xtrace:methods=[[[!]<method_spec>[[,...]]]
```

Where `<method_spec>` is formally defined as:

```
{[*][[*]<classname>[*].{[*][[*]<methodname>[*]]([)]
```

**Notes:**

- The exclamation point (!) in the `methods` parameter is a NOT operator. You can use this symbol and multiple methods in combination. For example, the following option traces all methods in the java.util.HashMap class except those beginning with put:
  
  `-Xtrace:methods={java/util/HashMap.*,!java/util/HashMap.put*},print=mt`

- The parentheses, (), that are in the `<method_spec>` variable define whether to trace method parameters. Method call parameters are traced only for interpreted methods. If the method was compiled by the JIT compiler, the parameters are not traced.

- If a method specification includes commas, the whole specification must be enclosed in braces:
  
  `-Xtrace:methods={java/lang/*,java/util/*},print=mt`

- You might have to enclose your command line in quotation marks. This action prevents the shell intercepting and fragmenting comma-separated command lines:

  "-Xtrace:methods={java/lang/*,java/util/*},print=mt"

Use the `print`, `maximal` and `output` options to route the trace to the required destination, where:

- `print` formats the trace point data while the Java application is running and writes the tracepoints to stderr.
- `maximal` saves the trace points into memory buffers.
- `output` writes the memory buffers to a file, in a binary compressed format.

To produce method trace that is routed to stderr, use the `print` option, specifying `mt` (method trace). For example: `-Xtrace:methods=java/lang/String.*,print=mt`. 
To produce method trace that is written to a binary file from the memory buffers, use the `maximal` and `output` options. For example: 

```
-Xtrace:methods=java/lang/String.*,maximal=mt,output=mytrace.trc
```

If you want your trace output to contain only the tracepoints you specify, use the option `-Xtrace:none` to switch off the default tracepoints. For example:

```
java -Xtrace:none -Xtrace:methods=java/lang/String.*,maximal=mt,output=mytrace.trc <class>
```

**Untraceable methods:**

Internal Native Library (INL) native methods inside the JVM cannot be traced because they are not implemented using JNI. The list of methods that are not traceable is subject to change without notice between releases.

The INL native methods in the JVM include:

- `java.lang.Class.allocateAndFillArray`
- `java.lang.Class.forNameImpl`
- `java.lang.Class.getClassDepth`
- `java.lang.Class.getClassLoaderImpl`
- `java.lang.Class.getComponentType`
- `java.lang.Class.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.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`

Chapter 10. Troubleshooting and support
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 = 000008B27D8
  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 = 000007FF7E7C450
  09:58:12.959 0x4236ce00 mt.18 - Instance method receiver: com/ibm/tools/attach/javaSE/AttachHandler@000007FF7E7C450 arguments: ((long)1000,(int)0)
  ```

  Method call parameters are traced only for interpreted methods. If the method was compiled by the JIT compiler, the parameters are not provided. For example:

  ```
  16:56:45.636 0x3e70000 mt.0 > java/lang/Thread.join(JI)V Bytecode method, This = 000007FF7E7C450
  16:56:45.648 0x3e70000 mt.18 - Instance method receiver: com/ibm/tools/attach/javaSE/AttachHandler@000007FF7E7C450 arguments: ((long)10000,(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
10:28:19.306 0x1a1100 mt.6 < java/lang/String.length()I
10:28:19.316 0x1a1100 mt.0 > java/util/HashMap.size()I
10:28:19.326 0x1a1100 mt.6 < java/util/HashMap.size()I

Tracing class constructors and static initializers:
-Xtrace:print=mt,methods=java/lang/Thread.<init>

This example traces the entry and exit of all constructors in the java.lang.Thread class.

13:08:19.937 0xf010500 mt.0 < java/lang/Thread.<init>()V
13:08:19.980 0xf010500 mt.6 < java/lang/Thread.<init>()V

-Xtrace:print=mt,methods=java/lang/Thread.<clinit>

This example traces the entry and exit of all static initializers in the java.lang.Thread class.

13:36:17.179 0xf010500 mt.3 < java/lang/Thread.<clinit>()V
13:36:17.182 0xf010500 mt.9 < java/lang/Thread.<clinit>()V

Note: These trace options are enclosed in quotation marks to avoid unwanted shell expansion of some of the characters.

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$Entry; Compile method, this = 0xc2540
10:37:42.246 0x1a1100 mt.6 < java/util/HashMap.createHashedEntry(Ljava/lang/Object;II)Ljava/util/HashMap$Entry; Compile method
10:37:42.256 0x1a1100 mt.1 > java/util/HashMap.findNonNullKeyEntry(Ljava/lang/Object;II)Ljava/util/HashMap$Entry; Compile method
10:37:42.266 0x1a1100 mt.7 < java/util/HashMap.findNonNullKeyEntry(Ljava/lang/Object;II)Ljava/util/HashMap$Entry; Compile method

Using triggers to trigger trace actions:
-Xtrace:print=mt,methods={java/io/PrintStream.println},trigger=method{java/io/PrintStream.println,jstacktrace}

This example triggers a stack trace when println is called:

14:48:27.056 0x1db7500 mt.0 < java/io/PrintStream.println(Ljava/lang/String;)V Bytecode method, This = fff0998
14:48:27.059 0x1db7500 j9trc_aux.0 - jstacktrace:
14:48:27.059 0x1db7500 j9trc_aux.1 - [1] java.io.PrintStream.println (PrintStream.java:818)
14:48:27.060 0x1db7500 mt.6 < java/io/PrintStream.println(Ljava/lang/String;)V Bytecode method
You can also use triggers to create dumps when methods are called. This example creates a Java dump when the constructor for the `java.lang.Thread` object is called, before the thread is started:

```
-Xtrace:print=mt,methods={java/lang/Thread.<init>},trigger=method{java/lang/Thread.<init>,javadump}
```

```
14:47:11.465*0xc8b500 mt.0 > java/lang/Thread.<init>(Ljava/lang/String;Ljava/lang/Object;IZ)V Bytecode method, This = fff02638
JVMDUMP034I User requested Java dump using 'javacore.20130619.154711.19206.0001.txt' through -Xtrace:trigger
```

See "trigger option" on page 414 for more information on using triggers.

Example of method trace output:

An example of method trace output.

Sample output using the command java -Xtrace:print=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.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.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.10 < 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.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.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.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
```

Chapter 10. Troubleshooting and support
The output lines comprise:

- 0x9e900, the current execenv (execution environment). Because every JVM thread has its own execenv, you can regard execenv as a thread-id. All trace with the same execenv relates to a single thread.
- The individual tracepoint ID in the mt component that collects and emits the data.
- The remaining fields show whether a method is being entered (>) or exited (<), followed by details of the method.

## JIT and AOT problem determination

You can use command-line options to help diagnose JIT and AOT compiler problems and to tune performance.

- “Diagnosing a JIT or AOT problem”
- “Performance of short-running applications” on page 435
- “JVM behavior during idle periods” on page 436

### 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”
- “Selectively disabling the JIT or AOT compiler” on page 431
- “Locating the failing method” on page 432
- “Identifying JIT compilation failures” on page 434
- “Identifying AOT compilation failures” on page 435

### Disabling the JIT or AOT compiler:

If you suspect that a problem is occurring in the JIT or AOT compiler, disable compilation to see if the problem remains. If the problem still occurs, you know that the compiler is not the cause of it.

### About this task

The JIT compiler is enabled by default. The AOT compiler is also enabled, but, is not active unless shared classes have been enabled. For efficiency reasons, not all methods in a Java application are compiled. The JVM maintains a call count for each method in the application; every time a method is called and interpreted, the call count for that method is incremented. When the count reaches the compilation threshold, the method is compiled and executed natively.

The call count mechanism spreads compilation of methods throughout the life of an application, giving higher priority to methods that are used most frequently.
Some infrequently used methods might never be compiled at all. As a result, when a Java program fails, the problem might be in the JIT or AOT compiler or it might be elsewhere in the JVM.

The first step in diagnosing the failure is to determine where the problem is. To do this, you must first run your Java program in purely interpreted mode (that is, with the JIT and AOT compilers disabled).

**Procedure**

1. Remove any -Xjit and -Xaot options (and accompanying parameters) from your command line.
2. Use the -Xint command-line option to disable the JIT and AOT compilers. For performance reasons, do not use the -Xint option in a production environment.

**What to do next**

Running the Java program with the compilation disabled leads to one of the following situations:

- The failure remains. The problem is not in the JIT or AOT compiler. In some cases, the program might start failing in a different manner; nevertheless, the problem is not related to the compiler.
- The failure disappears. The problem is most likely in the JIT or AOT compiler. If you are not using shared classes, the JIT compiler is at fault. If you are using shared classes, you must determine which compiler is at fault by running your application with only JIT compilation enabled. Run your application with the -Xnoaot option instead of the -Xint option. This leads to one of the following situations:
  - The failure remains. The problem is in the JIT compiler. You can also use the -Xnojit instead of the -Xnoaot option to ensure that only the JIT compiler is at fault.
  - The failure disappears. The problem is in the AOT compiler.

**Selectively disabling the JIT or AOT compiler:**

If your Java program failure points to a problem with the JIT or AOT compiler, you can try to narrow down the problem further.

**About this task**

By default, the JIT compiler optimizes methods at various optimization levels. Different selections of optimizations are applied to different methods, which are based on their call counts. Methods that are called more frequently are optimized at higher levels. By changing JIT compiler parameters, you can control the optimization level at which methods are optimized. You can determine whether the optimizer is at fault and, if it is, which optimization is problematic.

In contrast, the AOT compiler compiles methods only at the “cold” optimization level. Forcing the AOT compiler to compile a method at a higher level is not supported.

You specify JIT parameters as a comma-separated list, which is appended to the -Xjit option. The syntax is -Xjit:<param1>,<param2>=<value>. For example:

```
java -Xjit:verbose,optLevel=noOpt HelloWorld
```
runs the HelloWorld program, enables verbose output from the JIT, and makes the JIT generate native code without performing any optimizations. Optimization options are listed in “How the JIT compiler optimizes code” on page 106. The AOT compiler is controlled in a similar manner, by using the -xaot option. Use the -Xjit option when you are diagnosing JIT compiler problems, and the -xaot option when you are diagnosing AOT compiler problems.

Follow these steps to determine which part of the compiler is causing the failure:

Procedure
1. Set the JIT or AOT parameter count=0 to change the compilation threshold to zero. This parameter causes each Java method to be compiled before it is run. Use count=0 only when you are diagnosing problems, because a lot more methods are compiled, including methods that are used infrequently. The extra compilation uses more computing resources and slows down your application. With count=0, your application fails immediately when the problem area is reached. In some cases, by using count=1 can reproduce the failure more reliably.

2. Add disableInlining to the JIT or AOT compiler parameters. disableInlining disables the generation of larger and more complex code. If the problem no longer occurs, use disableInlining as a workaround while the Java service team analyzes and fixes the compiler problem.

3. Decrease the optimization levels by adding the optLevel parameter, and run the program again until the failure no longer occurs, or you reach the “noOpt” level. For a JIT compiler problem, start with “scorching” and work down the list. For an AOT compiler problem, start with “cold” and work down the list. The optimization levels are, in decreasing order:
   a. scorching
   b. veryHot
   c. hot
   d. warm
   e. cold
   f. noOpt

What to do next

If one of these settings causes your failure to disappear, you have a workaround that you can use. This workaround is temporary while the Java service team analyze and fix the compiler problem. If removing disableInlining from the JIT or AOT parameter list does not cause the failure to reappear, do so to improve performance. Follow the instructions in “Locating the failing method” to improve the performance of the workaround.

If the failure still occurs at the “noOpt” optimization level, you must disable the JIT or AOT compiler as a workaround.

Locating the failing method:

When you have determined the lowest optimization level at which the JIT or AOT compiler must compile methods to trigger the failure, you can find out which part of the Java program, when compiled, causes the failure. You can then instruct the compiler to limit the workaround to a specific method, class, or package, allowing the compiler to compile the rest of the program as usual. For JIT compiler failures,
if the failure occurs with `-Xjit:optLevel=noOpt`, you can also instruct the compiler to not compile the method or methods that are causing the failure at all.

**Before you begin**

If you see error output like this example, you can use it to identify the failing method:

```
Unhandled exception
Type=Segmentation error
vmState=0x00000000
Target=2_30_20050520_01866_BHdSMr (Linux 2.4.21-27.0.2.EL)
CPU=s390x (2 logical CPUs) (0x7b6a8000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=4148bf20 Signal_Code=00000001
Handler1=00000100002ADB14 Handler2=00000100002F480C InaccessibleAddress=0000000000000000
grpc0=0000000000000006 grpr1=0000000000000006 gpr2=0000000000000000 gpr3=0000000000000006
grpc4=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(I)(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 load)`. Methods for which AOT code is loaded from the shared class cache start with `+ (AOT load)`. 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.

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 `<n>` 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.

AOT methods can also be invalidated in the shared cache to prevent loading by using the `-Xshareclasses:invalidateAotMethods` suboption. For more information, see “-Xshareclasses” on page 560.

**Identifying JIT compilation failures:**

For JIT compiler failures, analyze the error output to determine if a failure occurs when the JIT compiler attempts to compile a method.

If the JVM crashes, and you can see that the failure has occurred in the JIT library (`libj9jit<vm_version>.so` or `libj9jit25.so`), the JIT compiler might have failed during an attempt to compile a method.
If you see error output like this example, you can use it to identify the failing method:

Unhandled exception
Type=Segmentation error
vmState=0x00050000
Target=2_30_20051215_04381_BHdSMr (Linux 2.4.21-32.0.1.EL)
CPU=ppc64 (4 logical CPUs) (0xebf4e000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=00000000 Signal_Code=00000001
Handler1=0000007FE05645B8 Handler2=0000007FE0615C20
R0=E8D4001870C00001 R1=0000007FF49181E0 R2=0000007FE2FBECE0 R3=0000007FF4E60D70
R4=E8D4001870C00000 R5=0000007FE2E02D30 R6=0000007FF4C0F188 R7=0000007FE2F8C290
Module=/home/test/sdk/jre/bin/libj9jit<vm_version>.so
Module_base_address=0000007FE29A6000

Method_being_compiled=com/sun/tools/javac/comp/Attr.visitMethodDef(Lcom/sun/tools/javac/tree/JCTree$JCMethodDecl)

The important lines are:

vmState=0x00050000
Indicates that the JIT compiler is compiling code. For a list of vmState code numbers, see the table in Javadump "TITLE, GPINFO, and ENVINFO sections" on page 343

Module=/home/test/sdk/jre/bin/libj9jit<vm_version>.so
Indicates that the error occurred in libj9jit<vm_version>.so, the JIT compiler module.

Method_being_compiled=
Indicates the Java method being compiled.

If your output does not indicate the failing method, use the verbose option with the following additional settings:
-Xjit:verbose={compileStart|compileEnd}

These verbose settings report when the JIT starts to compile a method, and when it ends. If the JIT fails on a particular method (that is, it starts compiling, but crashes before it can end), use the exclude parameter to exclude it from compilation (refer to “Locating the failing method” on page 432). 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 432, 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 431 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.

**IBM Support Assistant Data Collector**

IBM Support Assistant Data Collector, version 2.0 and later, helps you to quickly collect diagnostic files, such as dump, log, and configuration files. The tool also helps you to send the collected data to IBM, if required. You can download the tool for free, or use the online version.

The Java runtime environment produces multiple diagnostic files in response to events such as General Protection Faults, out of memory conditions or receiving unexpected operating system signals. Use the IBM Support Assistant Data Collector to collect this diagnostic data together, and send it to IBM if required.

There are different editions of the tool for various IBM products. For more information, and to access the 2.0 or later version for your product, see IBM Support Assistant Data Collectors.

**Note:** This tool replaces the Diagnostics Collector that was available in previous releases of the SDK and Runtime Environment. You can still specify the -Xdiagnosticscollector option on the command line, but this option only generates a message.

**Garbage Collector diagnostic data**

This section describes how to diagnose garbage collection.

The topics that are discussed in this chapter are:

- 
- 


"Verbose garbage collection logging"

"-Xtgc tracing" on page 452

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 69 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 577 for more information). If you redirect the output to a file, you can later analyze the file contents by using IBM Monitoring and Diagnostic Tools - Garbage Collection and Memory Visualizer. For more information about this tool, see "Using the IBM Monitoring and Diagnostic Tools" on page 319.

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 and a local timestamp. 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. These items can be modified with options such as -Xgcthreads.

The first tag shown in the output is the <initialized> tag, which is followed by values that include an id and timestamp. The information shown in the <initialized> section includes the garbage collection policy, the policy options, and any JVM command-line options that are in effect at the time.

<initialized id="1" timestamp="2010-11-23T00:41:32.328">
  <attribute name="gcPolicy" value="-Xgcpolicy:gencon" />
  <attribute name="maxHeapSize" value="0x5fcf0000" />
  <attribute name="initialHeapSize" value="0x400000" />
  <attribute name="compressedRefs" value="false" />
  <attribute name="pageSize" value="0x1000" />
  <attribute name="requestedPageSize" value="0x1000" />
  <attribute name="gcthreads" value="2" />
</system>
For more information about items in the garbage collection policy section, see the following pages:

- `-Xgcpolicy` option: “-Xgcpolicy” on page 584
- Initial and maximum heap sizes” on page 93
- Compressed references: “More effective heap usage using compressed references” on page 216
- Page sizes: “Configuring large page memory allocation” on page 231
- `-Xgcthreads` option: “-Xgcthreads” on page 585

For more information about system properties and command-line options, see “Command-line options” on page 525.

Stop-the-world operations:

When an application is stopped so that the garbage collector has exclusive access to the Java virtual machine verbose logging records the event.

The items in this section of the log are explained as follows:

`<exclusive-start>` and `<exclusive-end>`

These tags represent a stop-the-world operation. The tags have the following attributes:
timestamp
The local timestamp at the start or end of the stop-the-world operation.

<response-info>
This tag provides details about the process of acquiring exclusive access to the virtual machine. This tag has the following attributes:

timens The time, in milliseconds, that was taken to acquire exclusive access to the virtual machine. To obtain exclusive access, the garbage collection thread requests all other threads to stop processing, then waits for those threads to respond to the request. If this time is excessive, you can use the -Xdump:system:events command-line parameter to create a system dump. The dump file might help you to identify threads that are slow to respond to the exclusive access request. For example, the following option creates a system dump when a thread takes longer than one second to respond to an internal virtual machine request:
-Xdump:system:events=slow,filter=1000ms
For more information about creating dumps, see "Using dump agents" on page 321.

idlems 'Idle time' is the time between one of the threads responding and the final thread responding. During this time, the first thread is waiting, or 'idle'. The reported time for idlems is the mean idle time (in milliseconds) of all threads.

threads
The number of threads that were requested to release VM access. All threads must respond.

lastid The last thread to respond.

lastname
The name of the thread that is identified by the lastid attribute.

durationms
The total time for which exclusive access was held by the garbage collection thread.

Garbage collection cycle:

Verbose garbage collection output shows each garbage collection cycle enclosed within <cycle-start> and <cycle-end> tags. Each garbage collection cycle includes at least one garbage collection increment.

The <cycle-end> tag contains a context-id attribute that matches the id of the corresponding <cycle-start> tag.

In the example, the <cycle-end> tag has a context-id of 4, which reflects the id value that is shown for <cycle-start>.

The items in this section of the log are explained as follows:

<cycle-start> and <cycle-end>
These tags represent a garbage collection cycle. Each tag has the following attributes:
The type of garbage collection. This attribute can have the following values:

**scavenge**

Nursery collection is called a Scavenge.

**global**

Mark-sweep garbage collection on the entire heap, with an optional Compact pass. For more information about global garbage collection, see: "Detailed description of global garbage collection" on page 74.

**contextid**

The contextid attribute of the `<cycle-end>` tag matches the id attribute of the corresponding `<cycle-start>` tag. In the example, the value of 4 indicates that this `<cycle-end>` tag corresponds to the `<cycle-start id="4">` tag.

**timestamp**

The local timestamp at the time of the start or end of the garbage collection cycle.

**intervalms**

The amount of time, in milliseconds, since the start of the last collection of this type. For the `<cycle-start>` tag, this value therefore includes both the duration of the previous garbage collection cycle, and the interval between the end of the last collection cycle and the start of this collection cycle.

If you are using the balanced garbage collection policy, you might see the following line, which precedes the `<cycle-start>` tag:

```
<allocation-taxation id="28" taxation-threshold="2621440" timestamp="2014-02-17T16:21:44.325" intervalms="319.068">
```

This line indicates that the current garbage collection cycle was triggered due to meeting an allocation threshold that was set at the end of the previous cycle. The value of the threshold is reported.

**Garbage collection increment:**

A complete garbage collection increment is shown within `<gc-start>` and `<gc-end>` tags in the verbose output. Each garbage collection increment includes at least one garbage collection operation.
The following details can be found in the log:

<gc-start>
This tag represents the start of a garbage collection increment. This tag has the following attributes:

**type**
The type of garbage collection. This attribute can have the following values:

- **scavenge**
  Nursery collection is called a Scavenge.

- **global**
  Mark-sweep garbage collection on the entire heap, with an optional Compact pass. For more information about global garbage collection, see: "Detailed description of global garbage collection" on page 74

**contextid**
The contextid attribute matches the id attribute of the corresponding garbage collection cycle. In the example, the value of 4 indicates that this garbage collection increment is part of the garbage collection cycle that has the tag <cycle-start id="4">.

**timestamp**
The local time stamp at the start or end of the garbage collection increment.

The `<gc-start>` tag encloses a `<mem-info>` section, which provides information about the current state of the Java heap.

<gc-end>
This tag represents the end of a garbage collection increment. This tag has the following attributes:

**type**
The type of garbage collection. This attribute can have the following values:

- **scavenge**
  Nursery collection is called a Scavenge.

- **global**
  Mark-sweep garbage collection on the entire heap, with an optional Compact pass. For more information about global garbage collection, see: "Detailed description of global garbage collection" on page 74

**contextid**
The contextid attribute matches the id attribute of the corresponding garbage collection cycle. In the example, the value of 4 indicates that this garbage collection increment is part of the garbage collection cycle that has the tag <cycle-start id="4">.

**timestamp**
The local time stamp at the start or end of the garbage collection increment.

**usertimems**
The total time in CPU seconds that the garbage collection threads spent in user mode.

**systemtimems**
The total time in CPU seconds that the garbage collection threads spent in kernel mode. A high systemtimems value can suggest that there is a high overhead in work sharing between garbage collection processes.
collection threads. If this is the case, you can use the `-Xgcthreads` option to lower the garbage collection thread count.

**activeThreads**

The number of active garbage collection threads during this garbage collection increment. This number might be lower than the number of garbage collection threads reported when garbage collection is initialized.

The `<gc-end>` tag encloses a `<mem-info>` section, which provides information about the current state of the Java heap.

**<mem-info>**

This tag shows the cumulative amount of free space and total space in the Java heap, calculated by summing the nursery heap size and the tenure heap size. In earlier versions of the Java SDK, tilt ratio calculations were required if you used the Generational Concurrent Garbage Collector, because the total value did not account for survivor space in the nursery. From version 7 release 1, these calculations are no longer required.

\[
\text{reported-total-tenure-heap-size + reported-total-nursery-size/tilt-ratio}
\]

The tilt ratio is shown in the associated `<gc-op>` section of the garbage collection log.

**<mem>**

Within each `<mem-info>` tag, multiple `<mem>` tags show the division of available memory across the various memory areas. Each `<mem>` tag shows the amount of free space and total space that is used in that memory area before and after a garbage collection event. The free space is shown as a figure and as a rounded-down percentage. The memory area is identified by the `type` attribute, which has one of the following values:

**nursery**

If you are using the Generational Concurrent Garbage Collector, `nursery` indicates that this `<mem>` tag applies to the new area of the Java heap. For more information about the Generational Concurrent Garbage Collector, see: “Generational Concurrent Garbage Collector” on page 83.

**allocate**

Indicates that this `<mem>` tag applies to a space in the nursery where future allocations occur.

**survivor**

Indicates that this `<mem>` tag applies to a reserved space in the nursery that is used during the garbage collection cycle for moving survived objects from allocate that have not yet reached tenure age. The `survivor` space will always show 0% free when a Java application is running. This behavior occurs because the space is empty, but reserved.

**tenure**

Indicates that this `<mem>` tag applies to the tenure area, where objects are stored after they reach the tenure age. This memory type is further divided into `soa` and `loa` areas.

**soa**

Indicates that this `<mem>` tag applies to the small object area of the tenure space. This area is used for the first allocation attempt for an object.

**loa**

Indicates that this `<mem>` tag applies to the large object area.
of the tenure space. This area is used to satisfy allocations for large objects. For more information, see "Large Object Area" on page 72.

The <gc-end> tag also contains information about <pending-finalizers>. For more information and examples, see "Information about finalization" on page 449.

**Related information:**

"Tilt ratio" on page 84

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.

**Garbage collection operation:**

Every garbage collection increment contains at least one garbage collection operation, which is shown in the verbose output with a <gc-op> tag.

The <gc-op> output contains subsections that describe operations that are specific to the type of garbage collection operation. These subsections might change from release to release, when improvements are made to the technology or when new data becomes available.

The following log excerpt shows an example of a garbage collection operation:

```
<gc-op id="7" type="scavenge" timems="1.127" contextid="4" timestamp="2010-11-23T00:41:32.515">
...  
... subsections that are determined by the operation type
</gc-op>
```

The items in this section of the log are explained as follows:

**<gc-op>**

This tag represents a garbage collection operation, and has the following attributes:

- **type** The type of garbage collection. The value of this attribute depends on the stage of the garbage collection cycle and the garbage collection policy that is in use. The value determines the subsections that appear within the <gc-op> tag, as described later in this topic.

**Global garbage collection**

The following type values can occur during any part of a global garbage collection cycle, and with the following garbage collection policies: generational concurrent (gencon), optimize for throughput (optthruput), and optimize for pause time (optavgpause).

- **mark** The mark phase of garbage collection. During this phase, the garbage collector marks all the live objects. See "Subsections for mark operations" on page 445.

- **sweep** The sweep phase of garbage collection. During this phase, the garbage collector identifies the unused parts of the heap, avoiding the marked objects. See "Subsections for sweep operations" on page 446.
compact
The compact phase of garbage collection. During this phase, the garbage collector moves objects to create larger, unfragmented areas of free memory. The garbage collector also changes references to moved objects to point to the new object location. Operations of this type might not occur because compaction is not always required. See “Subsections for compact operations” on page 446.

classunload
The garbage collector unloads classes and class loaders that have no live object instances. Operations of this type might not occur because class unloading is not always required. See “Subsections for classunload operations” on page 447.

Final stop-the-world part of a concurrent global garbage collection
The following type values can occur only in the final stop-the-world part of a concurrent global garbage collection cycle, and with the following garbage collection policies: gencon and optavgpause. These operations occur before mandatory mark-sweep operations, in the order shown.

tracing
The garbage collector traces and marks live objects before the final card-cleaning phase. Operations of this type occur only if the concurrent phase of the global garbage collection cycle is abnormally halted. Normally, tracing and marking is done concurrently.

rs-scan
The garbage collector traces and marks objects in the nursery that were found through the remembered set. The remembered set is a list of objects in the old (tenured) heap that have references to objects in the new area.

card-cleaning
The final card-cleaning phase before final stop-the-world marking. This phase is a normal step in incremental-update concurrent marking. This phase compensates for live object mutation during concurrent tracing and marking.

gencon garbage collection
The following type value applies only to the gencon garbage collection policy, and occurs during local and nursery collections.

scavenge
A scavenger operation involves tracing live nursery objects and moving them to the survivor area. The operation also includes fixing or adjusting object
references for the whole heap. See “Subsections for scavenge operations” on page 447.

timems  The time, in milliseconds, taken to complete the garbage collection operation.

contextid  The contextid attribute matches the id attribute of the corresponding garbage collection cycle. In the example, the value of 4 indicates that this garbage collection increment is part of the garbage collection cycle that has the tag <cycle-start id="4">.

timestamp  The local time stamp at the time of the garbage collection operation.

The following information describes the subsections of the <gc-op> tag, which vary depending on the value of the <gc-op> type attribute.

Subsections for mark operations

The following log excerpt shows an example of a mark operation:

```
<gc-op id="9016" type="mark" timems="14.563" contextid="9013" timestamp="2015-09-28T14:47:49.927">
  <finalization candidates="1074" enqueued="11"/>
  <references type="soft" candidates="16960" cleared="10" enqueued="6" dynamicThreshold="12" maxThreshold="32"/>
  <references type="weak" candidates="6514" cleared="1" enqueued="1"/>
  <references type="phantom" candidates="92" cleared="0" enqueued="0"/>
  <stringconstants candidates="18027" cleared="1"/>
</gc-op>
```

The subsections within the <gc-op> tag are explained as follows:

<trace-info>  Contains general information about the objects traced. This tag has the following attributes:

objectcount  The number of objects discovered during the stop-the-world (STW) phase of marking.

scancount  The number of objects that are non-leaf objects: that is, they have at least one reference slot.

scanbytes  The total size in bytes of all scannable objects. (This is less than the total size of all live objects, the "live set.")

<finalization>  <references>  For information about the <finalization> and <references> elements, see “Information about finalization” on page 449 and “Information about reference processing” on page 450.

<stringconstants>  Contains general information about the objects traced. This tag has the following attributes:

candidates  The total number of string constants.
The number of string constants removed during this garbage collection cycle. (The number of string constants added since the previous global garbage collection is not explicitly reported.)

Subsections for sweep operations

The following log excerpt shows an example of a sweep operation:
<gc-op id="8979" type="sweep" timems="1.468" contextid="8974" timestamp="2015-09-28T14:47:49.141" />

There are no subsections within the <gc-op> tag.

Subsections for compact operations

The following log excerpt shows an example of a compact operation:
<gc-op id="8981" type="compact" timems="43.088" contextid="8974" timestamp="2015-09-28T14:47:49.184">
<compact-info movecount="248853" movebytes="10614296" reason="compact on aggressive collection" />
</gc-op>

There is one subsection within the <gc-op> tag:

<compact-info>
This tag has the following attributes:

movecount
The number of objects moved.

movebytes
The size of the objects moved in bytes.

reason
The reason for the compact operation:

compact to meet allocation
Unable to satisfy allocation even after mark-sweep.

compact on aggressive collection
Aggressive garbage collection is a global garbage collection that involves extra steps and gc-op operations to free as much memory as possible. One of those operations is compaction. Aggressive garbage collection might be triggered after a normal (non-aggressive) Global garbage collection was unable to satisfy the allocate operation. Note that alternate Explicit garbage collections (for example, those invoked with System.gc()) are aggressive.

heap fragmented
Compaction to reduce fragmentation, as measured by internal metrics. There are a number of reasons to reduce fragmentation such as to prevent premature allocation failures with large objects, increase locality of objects and references, lower contention in allocation, or reduce frequency of Global garbage collections.

forced gc with compaction
A Global garbage collection that included a compact operation was explicitly requested, typically by using an agent or RAS tool, before a heap dump, for example.

low free space
An indication that free memory is less than 4%.
very low free space
An indication that free memory is less than 128 kB.

forced compaction
Compaction was explicitly requested with one of the JVM options such as -Xcompactgc.

compact to aid heap contraction
Objects were moved in the heap from high to low address ranges to create contiguous free space and thus aid heap contraction.

Subsections for classunload operations

The garbage collector unloads classes and class loaders that have no live object instances. Operations of this type might not occur because class unloading is not always required. The following log excerpt shows an example of a classunload operation:

```xml
<gc-op id="8978" type="classunload" timems="1.452" contextid="8974" timestamp="2015-09-28T14:47:49.140">  
  <classunload-info classloadercandidates="1147" classloadersunloaded="3" classesunloaded="5" anonymousclassesunloaded="0" quiescems="0.000" setupms="1.408" scanms="0.041" postms="0.003"/>
</gc-op>
```

There is one subsection within the <gc-op> tag:

```xml
<classunload-info>
  This tag has the following attributes:
  
  classloadercandidates
  The total number of class loaders.
  
  classloadersunloaded
  The number of class loaders unloaded in this garbage collection cycle.
  
  classesunloaded
  The number of classes unloaded.
  
  anonymousclassesunloaded
  The number of anonymous classes unloaded. (Anonymous classes are unloaded individually and are reported separately.)
  
  quiescems
  setupms
  scanms
  postms
  The total time (in milliseconds) is broken down into four substeps.
</classunload-info>
```

Subsections for scavenge operations

Scavenge operations occur only with the gencon garbage collection policy. A scavenge operation runs when the allocate space within the nursery area is filled. During a scavenge, reachable objects are copied either into the survivor space within the nursery, or into the tenure space if they have reached the tenure age. For more information, see “Generational Concurrent Garbage Collector” on page 83.

The following log excerpt shows an example of a scavenge operation:

```xml
<gc-op id="9029" type="scavenge" timems="2.723" contextid="9026" timestamp="2015-09-28T14:47:49.998">  
  <scavenger-info tenureage="3" tenuremask="ffb8" tiltratio="89"/>
  <memory-copied type="nursery" objects="11738" bytes="728224" bytesdiscarded="291776"/>
  <memory-copied type="tenure" objects="6043" bytes="417920" bytesdiscarded="969872"/>
</gc-op>
```
The subsections within the <gc-op> tag are explained as follows:

<scavenger-info>
Contains general information about the operation. This tag has the following attributes:

tenureage
The current age at which objects are promoted to the tenure area. For more information, see "Tenure age" on page 84.

tenuremask
tiltratio
The tilt ratio (a percentage) after the last scavenge event and space adjustment. The scavenger redistributes memory between the allocate and survivor areas by using a process called "tilting". Tilting controls the relative sizes of the allocate and survivor spaces, and the tilt ratio is adjusted to maximize the amount of time between scavenges. A tilt ratio of 60% indicates that 60% of new space is reserved for allocate space and 40% for survivor space. For more information, see "Tilt ratio" on page 84.

<memory-copied>
Indicates the quantity of object data that is flipped to the nursery area or promoted to the tenure area. This tag has the following attributes:

type One of the values nursery or tenure.

objects
The number of objects flipped to the nursery area or promoted to the tenure area.

bytes
The number of bytes flipped to the nursery area or promoted to the tenure area.

bytesdiscarded
The number of bytes consumed in the nursery or tenure area but not successfully used for flipping or promotion. For each area, the total amount of consumed memory is the sum of the values of bytes and bytesdiscarded.

<finalization>
<references>
For information about the <finalization> and <references> elements, see “Information about finalization” on page 449 and “Information about reference processing” on page 450.

Related information:
"Detailed description of global garbage collection” on page 74

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.
Information about finalization:

The <finalization> section of the log records the number of enqueued finalizable objects that are in the current GC operation. The <pending-finalizers> section, which is found in the <gc-end> tag, records the current pending state. The current pending state is the sum of the enqueued finalizable objects from the current GC operation, plus all the objects from the past that are not yet finalized.

The following log excerpt shows an example of a <finalization> entry in the log:
<finalization candidates="1088" enqueued="10">
  <finalization>
  This tag shows the number of objects that contain finalizers and were queued for virtual machine finalization during the collection. This number is not equal to the number of finalizers that were run during the collection because finalizers are scheduled by the virtual machine. This tag has the following attributes:
  
  candidates
  Indicates the number of finalizable objects that were found in the GC cycle. The number includes live finalizable objects and those finalizable objects that are no longer alive since the last GC cycle. Only those objects that are no longer alive are enqueued for finalization.

  enqueued
  Indicates the fraction of candidates that are eligible for finalization.

The following log excerpt shows an example of a <pending-finalizers> entry in the log, which is recorded only in the gc-end section.
<pending-finalizers system="3" default="7" reference="40" classloader="0" />

<pending-finalizers>
  Indicates the current state of queues of finalizable objects.

  system
  Indicates the number of enqueued system objects.

  default
  Indicates the number of enqueued default objects. At the end of the GC cycle, the sum of system objects and default objects is larger than or equal to the fraction of candidates that are eligible for finalization from the same GC cycle.

  reference
  Indicates the number of enqueued references. That is, the number of references that were cleared and have a reference queue that is associated with them since the previous GC cycle. Typically, the number of pending references is larger or equal to the sum of enqueued weak, soft, and phantom references reported in the gc-op stanza of the same cycle.

  classloader
  Indicates the number of class loaders that are eligible for asynchronous unloading.

If the number of pending finalizers is larger than the number of candidates that are created by the current GC cycle, finalization cannot keep up with the influx. This situation might indicate suboptimal behavior. You can work out the number of outstanding pending finalizer objects at the beginning of a GC cycle by using the following calculation:
number of outstanding pending finalizer objects at the beginning =
number of pending finalizer objects at the end - candidates created in this cycle

Information about reference processing:

The <references> section in the verbose Garbage Collection (GC) logs contains
information about reference processing.

The following log excerpt shows an example of a <references> entry in the log:

<references type="soft" candidates="16778" cleared="21" enqueued="14" dynamicThreshold="10" maxThreshold="32" />
<references type="weak" candidates="5916" cleared="33" enqueued="26" />

This tag provides information about Java reference objects, and has the
following attributes:

- **type** Indicates the type of the reference object. The type affects how the
  reference object is processed during garbage collection, as
described in “Soft, weak, and phantom reference processing” on
  page 80. The type attribute can have the following values:
  - **soft** Indicates that this object is an instance of the SoftReference class. Soft references are processed first during garbage
collection.
  - **weak** Indicates that this object is an instance of the
    WeakReference class. Weak references are processed after
    soft references during garbage collection.
  - **phantom** Indicates that this object is an instance of the
    PhantomReference class. Phantom references are processed
    after weak references during garbage collection.

For more information about references, see “Reference objects” on
page 80.

- **candidates** Indicates the number of reference objects that were found in the
  GC cycle. The number includes reference objects whose referents
  are strong, soft, weak, or phantom reachable.

- **cleared** Indicates the number of reference objects that have a soft, weak, or
  phantom reachable referent, and that are cleared in this GC cycle.

- **enqueued** Indicates the fraction of cleared reference objects that are eligible
  for enqueuing. Eligible objects are cleared references objects that
  have a ReferenceQueue associated with them at reference creation
time. The reference enqueuing is done by the finalization thread.
For more information about the finalization section of the log, see
“Information about finalization” on page 449.

- **dynamicThreshold** Applicable only to soft reference types. Indicates a dynamic value
  for the number of GC cycles (including local or global GC cycles)
  that a soft reference object can survive before it is cleared. The
dynamic number is generated by internal heuristics that can reduce
the threshold. For example, high heap occupancy might reduce the
threshold from the maximum value.
maxThreshold
Applicable only to soft reference types. This value shows the maximum number of GC cycles (including local or global) that a soft reference object can survive before it is cleared.

Allocation failure:

Garbage collection cycles caused by an allocation failure are shown by <af-start> and <af-end> tags in the verbose output.

The <af-start> and <af-end> tags enclose the <cycle-start> and <cycle-end> tags. The <af-start> tag contains a totalBytesRequested attribute. This attribute specifies the number of bytes that were required by the allocations that caused this allocation failure. The intervals attribute on the af-start tag is the time, in milliseconds, since the previous <af-start> tag. When the garbage collection cycle caused by the allocation failure is complete, an allocation-satisfied tag is generated. This tag indicates that the allocation that caused the failure is now complete.

The following example shows two cycles within an af-start/af-end pair. Typically there is only one cycle, but in this example, Scavenge is not able to release enough memory in either Nursery or Tenure to meet the value of totalBytesRequested. This failure triggers global garbage collection, after which the allocation request is fulfilled.

The items in this section of the log are explained as follows:

<af-start> and <af-end>
This tag is generated when an allocation failure occurs, and contains a garbage collection cycle, indicated by the <cycle-start> and <cycle-end> tags. This tag has the following attributes:

totalBytesRequested
The number of bytes that were required by the allocations that caused this allocation failure.

timestamp
The local timestamp at the time of the allocation failure.

intervals
The time, in milliseconds, since the previous <af-start> tag was generated.

<allocation-satisfied>
This tag indicates that the allocation that caused the failure is complete. This tag is generated when the garbage collection cycle that was caused by the allocation failure is complete. This tag has the following attributes:

thread
The Java thread identifier that triggers garbage collection.
This attribute is identical to the totalBytesRequested attribute that is seen in the `<af-start>` tag.

-Xtg<trace

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 -Xtg<trace available. The output is written to stdout. More than one trace can be enabled simultaneously by separating the parameters with commas, for example `-Xtg:backtrace,compaction`.

-Xtg<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.

-Xtg<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.

-Xtg<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.
This trace shows when the background thread was activated, and the amount of tracing it performed (in bytes).

-Xtgcdump:

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:

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.

-XtgceXcessiveGC:

This trace shows statistics for garbage collection cycles.

After a garbage collection cycle has completed, a trace entry is produced:

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" \n  activesize="16777216" currentsize="16777216" maximumsize="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 577).

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).

-Xtgcsavenger:

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 scavenger is shown as follows:

(SCAV: tgcScavenger OBJECT HISTOGRAM)

(SCAV: class | instances of age 0-14 in semi-space |
SCAV: java/lang/ref/SoftReference 0 0 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/PrintStream 0 1 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/InputStream 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
SCAV: java/lang/StringBuffer 0 4 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: java/io/OutputStream 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
SCAV: java/lang/StringBuffer 0 4 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/CharToByteCp1252 0 1 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

-Xtgcterse:

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)* 230A0778 a x0000001C java/lang/String
*DH(1)* 230A0794 a x00000048 char[]
*DH(1)* 230A07DC a x00000118 java/lang/StringBuffer
*DH(1)* 230A07F4 a x00000030 char[]
*DH(1)* 230A0824 a x00000054 char[]
*DH(1)* 230A0878 a x0000001C java/lang/String
*DH(1)* 230A0994 a x00000018 java/util/HashMapEntry
*DH(1)* 230A08AC a x0000004C char[]
*DH(1)* 230A08F8 a x0000001C java/lang/String
*DH(1)* 230A0914 a x0000004C char[]
*DH(1)* 230A0960 a x0000001B char[]
*DH(1)* 230A0978 a x0000001C java/lang/String
*DH(1)* 230A0994 a x00000018 char[]
*DH(1)* 230A09AC a x0000001B java/lang/StringBuffer
*DH(1)* 230A09C4 a x00000030 char[]
*DH(1)* 230A09F4 a x00000054 char[]
*DH(1)* 230ADA48 a x0000001C java/lang/String
Troubleshooting the Metronome Garbage Collector

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

Using verbose:gc information:

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

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

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

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

Example of verbose:gc output

Enter:

```
java -Xgcpolicy:metronome -verbose:gc -Xgc:verboseGCCycleTime=N myApplication
```

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

```
<trigger-start id="25" timestamp="2011-07-12T09:32:04.503" />
<cycle-start id="26" type="global" contextid="26" timestamp="2011-07-12T09:32:04.503" intervalms="984.285" />
<gc-op id="27" type="heartbeat" contextid="26" timestamp="2011-07-12T09:32:05.209">
  <quanta quantumCount="321" quantumType="mark" minTimeMs="0.367" meanTimeMs="0.524" maxTimeMs="1.878" maxTimestampMs="598704.070" />
  <exclusiveaccess-info minTimeMs="0.006" meanTimeMs="0.062" maxTimeMs="0.147" />
</gc-op>
```
<free-mem type="heap" minBytes="99143592" meanBytes="114374153" maxBytes="134182032" />
<thread-priority maxPriority="11" minPriority="11" />
</gc-op>

<gc-op id="28" type="heartbeat" contextid="26" timestamp="2011-07-12T09:32:05.458">
  <quanta quantumCount="115" quantumType="sweep" minTimeMs="0.430" meanTimeMs="0.471" maxTimeMs="0.511"
    maxTimestampMs="599475.654" />
  <exclusiveaccess-info minTimeMs="0.007" meanTimeMs="0.067" maxTimeMs="0.173" />
  <classunload-info classloadersunloaded=9 classesunloaded=156 />
  <references type="weak" cleared="660" />
</gc-op>

<gc-op id="29" type="syncgc" timems="136.945" contextid="26" timestamp="2011-07-12T09:32:06.046">
  <syncgc-info reason="out of memory" exclusiveaccessTimeMs="0.006" threadPriority="11" />
  <free-mem-delta type="heap" bytesBefore="21290752" bytesAfter="171963656" />
</gc-op>

<cycle-end id="30" type="global" contextid="26" timestamp="2011-07-12T09:32:06.046" />
<trigger-end id="31" timestamp="2011-07-12T09:32:06.046" />

The following event types can occur:

<trigger-start ...
  The start of a garbage collection cycle, when the amount of used memory became higher than the trigger threshold. The default threshold is 50% of the heap. The intervalsms attribute is the interval between the previous trigger end event (with id-1) and this trigger start event.

<trigger-end ...
  A garbage collection cycle successfully lowered the amount of used memory beneath the trigger threshold. If a garbage collection cycle ended, but used memory did not drop beneath the trigger threshold, a new garbage collection cycle is started with the same context ID. For each trigger start event, there is a matching trigger end event with same context ID. The intervalsms attribute is the interval between the previous trigger start event and the current trigger end event. During this time, one or more garbage collection cycles will have completed until used memory has dropped beneath the trigger threshold.

<gc-op id="28" type="heartbeat"...>
  A periodic event that gathers information (on memory and time) about all garbage collection quanta for the time covered. A heartbeat event can occur only between a matching pair of trigger start and trigger end events; that is, while an active garbage collection cycle is in process. The intervalsms attribute is the interval between the previous heartbeat event (with id -1) and this heartbeat event.

<gc-op id="29" type="syncgc"...>
  A synchronous (nondeterministic) garbage collection event. See “Synchronous garbage collections” on page 458

The XML tags in this example have the following meanings:

<quanta ...
  A summary of quantum pause times during the heartbeat interval, including the length of the pauses in milliseconds.

<free-mem type="heap" ...
  A summary of the amount of free heap space during the heartbeat interval, sampled at the end of each garbage collection quantum.
The number of classloaders and classes unloaded during the heartbeat interval.

The number and type of Java reference objects that were cleared during the heartbeat interval.

Note:

- If only one garbage collection quantum occurred in the interval between two heartbeats, the free memory is sampled only at the end of this one quantum. Therefore the minimum, maximum, and mean amounts given in the heartbeat summary are all equal.

- The interval between two heartbeat events might be significantly larger than the cycle time specified if the heap is not full enough to require garbage collection activity. For example, if your program requires garbage collection activity only once every few seconds, you are likely to see a heartbeat only once every few seconds.

- It is possible that the interval might be significantly larger than the cycle time specified because the garbage collection has no work on a heap that is not full enough to warrant garbage collection activity. For example, if your program requires garbage collection activity only once every few seconds, you are likely to see a heartbeat only once every few seconds.

If an event such as a synchronous garbage collection or a priority change occurs, the details of the event and any pending events, such as heartbeats, are immediately produced as output.

- If the maximum garbage collection quantum for a given period is too large, you might want to reduce the target utilization using the -Xgc:targetUtilization option. This action gives the Garbage Collector more time to work. Alternatively, you might want to increase the heap size with the -Xmx option. Similarly, if your application can tolerate longer delays than are currently being reported, you can increase the target utilization or decrease the heap size.

- The output can be redirected to a log file instead of the console with the -Xverbosegclog:<file> option; for example, -Xverbosegclog:out writes the -verbose:gc output to the file out.

- The priority listed in thread-priority is the underlying operating system thread priority, not a Java thread priority.

Synchronous garbage collections

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

- An explicit System.gc() call in the code.
- The JVM runs out of memory then performs a synchronous garbage collection to avoid an OutOfMemoryError condition.
- The JVM shuts down during a continuous garbage collection. The JVM cannot cancel the collection, so it completes the collection synchronously, and then exits.

An example of a System.gc() entry is:
<gc-op id="9" type="syncgc" timems="12.92" contextid="8" timestamp= "2011-07-12T09:41:40.808"> <syncgc-info reason="system GC" totalBytesRequested="260" exclusiveaccessTimeMs="0.009" threadPriority="11" /> <free-mem-delta type="heap" bytesBefore="22085440" bytesAfter="136023450" /> <classunload-info classloadersunloaded="54" classesunloaded="234" />
An example of a synchronous garbage collection entry as a result of the JVM shutting down is:

```xml
<gc-op id="24" type="syncgc" timems="6.439" contextid="19" timestamp="2011-07-12T09:43:14.524">  
  <syncgc-info reason="VM shut down" exclusiveaccessTimeMs="0.009" threadPriority="11" />  
  <free-mem-delta type="heap" bytesBefore="56182430" bytesAfter="151356238" />  
  <classunload-info classloadersunloaded="14" classesunloaded="276" />  
  <references type="soft" cleared="154" dynamicThreshold="29" maxThreshold="32" />  
  <references type="weak" cleared="53" />  
  <finalization enqueued="34" />  
</gc-op>
```

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

- `<gc-op id="9" type="syncgc" timems="6.439" ...>`
  - This line indicates that the event type is a synchronous garbage collection. The `timems` attribute is the duration of the synchronous garbage collection in milliseconds.

- `<syncgc-info reason="..."/>`
  - The cause of the synchronous garbage collection.

- `<free-mem-delta.../>`
  - The free Java heap memory before and after the synchronous garbage collection in bytes.

- `<finalization .../>`
  - The number of objects awaiting finalization.

- `<classunload-info .../>`
  - The number of classloaders and classes unloaded during the heartbeat interval.

- `<references type="weak" cleared="53" .../>`
  - The number and type of Java reference objects that were cleared during the heartbeat interval.

Synchronous garbage collection due to out-of-memory conditions or VM shutdown can happen only when the Garbage Collector is active. It has to be preceded by a trigger start event, although not necessarily immediately. Some heartbeat events probably occur between a trigger start event and the `syncgc` event. Synchronous garbage collection caused by `System.gc()` can happen at any time.

### Tracking all GC quanta

Individual GC quanta can be tracked by enabling the `GlobalGCStart` and `GlobalGCEnd` tracepoints. These tracepoints are produced at the beginning and end of all Metronome Garbage Collector activity including synchronous garbage collections. The output for these tracepoints looks similar to:

```
03:44:35.281 0x833cd00 j9mm.52 - GlobalGC start: globalcount=3
```

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

### Out-of-memory entries

When the heap runs out of free space, an entry is written to the `-verbose:gc` log before the OutOfMemoryError exception is thrown. An example of this output is:
By default a Javadoc dump is produced as a result of an OutOfMemoryError exception. This dump contains information about the memory used by your program.

```
NULL
1STSEGTOTAL  Total memory: 4066080  (0x003E0B20)
1STSEGINUSE  Total memory in use: 3919440  (0x003BCE50)
1STSEGFREE   Total memory free:  146640  (0x00023CD0)
```

Metronome Garbage Collector behavior in out-of-memory conditions:

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

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

**Related information:**

Using `-Xverbose:gc` to analyze synchronous garbage collections

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

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

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

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

```
<gc-op id="9" type="syncgc" timems="6,439" contextid="8" timestamp="2011-07-12T09:41:40.808">
  <syncgc-info reason="VM shut down" exclusiveaccessTimeMs="0.009" threadPriority="11"/>
  <free-mem-delta type="heap" bytesBefore="126082300" bytesAfter="156085440"/>
  <classunload-info classloadersunloaded="14" classesunloaded="276"/>
  <references type="soft" cleared="154" dynamicThreshold="29" maxThreshold="32"/>
  <references type="weak" cleared="53"/>
  <finalization enqueued="34"/>
</gc-op>
```

**Class-loader diagnostic data**

There is some diagnostic data that is available for class-loading.

The topics that are discussed in this chapter are:

- “Class-loader command-line options” on page 461
- “Class-loader runtime diagnostic data” on page 461
- “Loading from native code” on page 462
**Class-loader command-line options**

There are some extended command-line options that are available

These options are:

- **-verbose:dyload**
  Provides detailed information as each class is loaded by the JVM, including:
  - The class name and package.
  - For class files that were in a .jar file, the name and directory path of the .jar (for bootstrap classes only).
  - Details of the size of the class and the time taken to load the class.

  The data is written out to stderr. An example of the output follows:

  <Loaded java/lang/String from C:\sdk\jre\lib\vm.jar>
  <Class size 17258; ROM size 21080; debug size 0>
  <Read time 27368 usec; Load time 782 usec; Translate time 927 usec>

- **-Xfuture**
  Turns on strict class-file format checks. Use this flag when you are developing new code because stricter checks will become the default in future releases. By default, strict format checks are disabled.

- **-Xverify[:<option>]**
  With no parameters, enables the Java bytecode verifier, which is the default. Therefore, if used on its own with no parameters, the option has no effect. Optional parameters are:
  - all - enable maximum verification
  - none - disable the verifier
  - remote - enables strict class-loading checks on remotely loaded classes

  The verifier is on by default and must be enabled for all production servers. Running with the verifier off, is not a supported configuration. If you encounter problems and the verifier was turned off using **-Xverify:none**, remove this option and try to reproduce the problem.

**Class-loader runtime diagnostic data**

Use the command-line parameter **-Dibm.cl.verbose=<class_expression>** to enable you to trace the way the class loaders find and load application classes.

The <class_expression> can be given as any Java regular expression. “Dic***” matches all classes with names begins with “Dic”, and so on.

Alternatively, you can use [IBM Monitoring and Diagnostic Tools - Health Center](https://www.ibm.com/support/home) to monitor the application and view class loading information such as when the class was loaded, whether the class was loaded from the shared cache, the number of instances of the class, and the amount of heap space that those instances are occupying.

Here is an example of the **-Dibm.cl.verbose** parameter:

C:\j9test>java -Dibm.cl.verbose=*HelloWorld hw.HelloWorld

This example produces output that is similar to this:

ExtClassLoader attempting to find hw.HelloWorld
ExtClassLoader using classpath C:\sdk\jre\lib\ext\CmpCrmf.jar;C:\sdk\jre\lib\ext\dtfj-interface.jar;
C:\sdk\jre\lib\ext\dtfj.jar;C:\sdk\jre\lib\ext\gskikm.jar;C:\sdk\jre\lib\ext\ibmcmsprovider.jar;C:\sdk\jre\lib\ext\ibmjcefips.jar;C:\sdk\jre\lib\ext\ibmjcep.provider.jar;C:\sdk\jre\lib\ext\ibmkeycert.jar
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.

**Loading from native code**
A class loader loads native libraries for a class.

Class loaders look for native libraries in different places:

- If the class that makes the native call is loaded by the Bootstrap class loader, this loader looks in the path that is specified by the sun.boot.library.path property, to load the libraries.

- If the class that makes the native call is loaded by the Extensions class loader, this loader looks in the paths that are specified by the following properties, in this order:
  1. java.ext.dirs
  2. sun.boot.library.path
  3. java.library.path
• If the class that makes the native call is loaded by the Application class loader, this loader looks in the paths that are specified by the following properties, in this order:
  1. sun.boot.library.path
  2. java.library.path
• If the class that makes the native call is loaded by a Custom class loader, this loader defines the search path to load libraries.

**Shared classes diagnostic data**

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

For an introduction to shared classes, see "Class data sharing" on page 104.

The topics that are discussed in this chapter are:

- “Deploying shared classes”
- “Dealing with runtime bytecode modification” on page 471
- “Understanding dynamic updates” on page 475
- “Using the Java Helper API” on page 477
- “Understanding shared classes diagnostic output” on page 480
- “Debugging problems with shared classes” on page 487
- “Class sharing with OSGi ClassLoading framework” on page 491

**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. The `cachedir` suboption also affects the permissions of persistent caches. For more information about the `-Xshareclasses:cachedir=` and `cacheDirPerm` suboptions, see "Class data sharing command-line options" on page 241.

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 470 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 Java virtual machine 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. Users can reduce start time by opening the cache as read-only, which avoids any contention that might be experienced when users access the cache in read/write mode. 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.

Note: Service refresh 2 and later: To enable read-only access of persistent caches by users outside the group of the cache creator, you must specify a directory with appropriate permissions when you create the cache. Specify this directory by using the -Xshareclasses:cacheDir option. If you do not use the -Xshareclasses:cacheDir option, the cache is created with default permissions. The default permissions deny access for users that are not in the same group as the cache creator. For more information, see "Class data sharing command-line options" on page 241.

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 tries to remove 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 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.

Related reference:

"-Xshareclasses" on page 560

Enables class sharing. This option can take a number of suboptions, some of which are cache utilities.

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 477. 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 549 for more information.

In general, the default settings provide significant startup performance benefits and use only a small amount of cache space. In some cases, for example, running the JVM without the JIT, there is no benefit gained from the cached AOT code. In these cases, turn off caching of AOT code.

To diagnose AOT issues, use the `-Xshareclasses:verboseAOT` command-line option. This option generates messages when AOT code is found or stored in the cache.

Caching JIT data

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

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

In general, the default settings provide significant performance benefits and use only a small amount of cache space.

Making the most efficient use of cache space

A shared class cache is a finite size and cannot grow. The JVM makes more efficient use of cache space by sharing strings between classes, and ensuring that classes are not duplicated. However, there are also command-line options that optimize the cache space available.

- `-Xscminaot` and `-Xscmaxaot` place maximum and minimum limits on the amount of AOT data the JVM can store in the cache. `-Xshareclasses:noaot` prevents the JVM from storing any AOT data.

- `-Xscminjitdata<size>` and `-Xscmaxjitdata<size>` place maximum and minimum limits on the amount of JIT data the JVM can store in the cache. `-Xshareclasses:nojitdata` prevents the JVM from storing any JIT data.

- `-Xshareclasses:nobootclasspath` disables the sharing of classes on the boot classpath, so that only classes from application class loaders are shared. There are also optional filters that can be applied to Java classloaders to place custom limits on the classes that are added to the cache.
Very long classpaths

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

Matching very long classpaths is initially expensive, but successful and failed matches are remembered. Therefore, loading classes from the cache using very long classpaths is much faster than loading from disk.

Growing classpaths

Where possible, avoid gradually growing a classpath in a URLClassLoader using addURL(). Each time an entry is added, an entire new classpath must be added to the cache.

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

Concurrent access

A shared class cache can be updated and read concurrently by any number of JVMs. Any number of JVMs can read from the cache while a single JVM is writing to it.

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

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

Class GC with shared classes

Running with shared classes has no affect on class garbage collection. Class loaders loading classes from the shared class cache can be garbage collected in the same way as class loaders that load classes from disk. If a class loader is garbage collected, the ROMClasses it has added to the cache persist.

Class Debug Area

A portion of the shared classes cache is reserved for storing the class attribute information LineNumberTable and LocalVariableTable during JVM debugging. By storing these attributes in a separate region, the operating system can decide whether to keep the region in memory or on disk, depending on whether debugging is taking place.

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

- -Xscdmx1048576
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

JVMPORTO20W 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 run time is an increasingly popular way to engineer required function into classes. Sharing modified bytecode improves startup time, especially when the modification being used is expensive. You can safely cache modified bytecode and share it between JVMs, but there are many potential problems because of the added complexity. It is important to understand the features described in this section to avoid any potential problems.

This section contains a brief summary of the tools that can help you to share modified bytecode.
Potential problems with runtime bytecode modification:

The sharing of modified bytecode can cause potential problems.

When a class is stored in the cache, the location from which it was loaded and a time stamp indicating version information are also stored. When retrieving a class from the cache, the location from which it was loaded and the time stamp of that location are used to determine whether the class should be returned. The cache does not note whether the bytes being stored were modified before they were defined unless it is specifically told so. Do not underestimate the potential problems that this modification could introduce:

- In theory, unless all JVMs sharing the same classes are using exactly the same bytecode modification, JVMs could load incorrect bytecode from the cache. For example, if JVM1 populates a cache with modified classes and JVM2 is not using a bytecode modification agent, but is sharing classes with the same cache, it could incorrectly load the modified classes. Likewise, if two JVMs start at the same time using different modification agents, a mix of classes could be stored and both JVMs will either throw an error or demonstrate undefined behavior.

- An important prerequisite for caching modified classes is that the modifications performed must be deterministic and final. In other words, an agent which performs a particular modification under one set of circumstances and a different modification under another set of circumstances, cannot use class caching. This is because only one version of the modified class can be cached for any given agent and once it is cached, it cannot be modified further or returned to its unmodified state.

In practice, modified bytecode can be shared safely if the following criteria are met:

- Modifications made are deterministic and final (described previously).
- The cache knows that the classes being stored are modified in a particular way and can partition them accordingly.

The VM provides features that allow you to share modified bytecode safely, for example using "modification contexts". However, if a JVMTI agent is unintentionally being used with shared classes without a modification context, this usage does not cause unexpected problems. In this situation, if the VM detects the presence of a JVMTI agent that has registered to modify class bytes, it forces all bytecode to be loaded from disk and this bytecode is then modified by the agent. The potentially modified bytecode is passed to the cache and the bytes are compared with known classes of the same name. If a matching class is found, it is reused; otherwise, the potentially modified class is stored in such a way that other JVMs cannot load it accidentally. This method of storing provides a "safety net" that ensures that the correct bytecode is always loaded by the JVM running the agent, but any other JVMs sharing the cache will be unaffected. Performance during class loading could be affected because of the amount of checking involved, and because bytecode must always be loaded from disk. Therefore, if modified bytecode is being intentionally shared, the use of modification contexts is recommended.

Modification contexts:

A modification context creates a private area in the cache for a given context, so that multiple copies or versions of the same class from the same location can be stored using different modification contexts. You choose the name for a context, but it must be consistent with other JVMs using the same modifications.
For example, one JVM uses a JVMTI agent "agent1", a second JVM uses no bytecode modification, a third JVM also uses "agent1", and a fourth JVM uses a different agent, "agent2". If the JVMs are started using the following command lines (assuming that the modifications are predictable as described previously), they should all be able to share the same cache:

```
java -agentlib:agent1 -Xshareclasses:name=cach1,modified=myAgent1 myApp.ClassName
java -Xshareclasses:name=cach1 myApp.ClassName
java -agentlib:agent1 -Xshareclasses:name=cach1,modified=myAgent1 myApp.ClassName
java -agentlib:agent2 -Xshareclasses:name=cach1,modified=myAgent2 myApp.ClassName
```

**SharedClassHelper partitions:**

Modification contexts cause all classes loaded by a particular JVM to be stored in a separate cache area. If you need a more granular approach, the SharedClassHelper API can store individual classes under "partitions".

This ability to use partitions allows an application class loader to have complete control over the versioning of different classes and is particularly useful for storing bytecode woven by Aspects. A partition is a string key used to identify a set of classes. For example, a system might weave a number of classes using a particular Aspect path and another system might weave those classes using a different Aspect path. If a unique partition name is computed for the different Aspect paths, the classes can be stored and retrieved under those partition names.

The default application class loader or bootstrap class loader does not support the use of partitions; instead, a SharedClassHelper must be used with a custom class loader.

**Using the JVMTI ClassFileLoadHook with cached classes:**

The `-Xshareclasses:enableBCI` suboption improves startup performance without using a modification context, when using JVMTI class modification. This suboption allows classes loaded from the shared cache to be modified using a JVMTI ClassFileLoadHook, or a java.lang.instrument agent, and prevents modified classes being stored in the shared classes cache.

Modification contexts allow classes modified at run time by JVMTI agents to be stored, logically separated, in the cache. This separation prevents conflicts with versions of the same class that are being used by other JVMs connected to the cache. However, there are a number of issues:

- Loading classes from the cache does not generate a callback to the JVMTI ClassFileLoadHook event, which prevents a JVMTI agent making any subsequent modifications. The ClassFileLoadHook event expects original class data to be passed back. This data is typically not available in the shared cache unless the cache was created with a JVMTI agent that is retransformation capable. This constraint might be undesirable for JVMTI or java.lang.instrument agents that want the ClassFileLoadHook event to be triggered every time, whether the class is loaded from the cache, or from the disk.
- If the JVMTI agent applies different runtime modifications every time the application is run, there will be multiple versions of the same class in the cache that cannot be reused or shared across JVMs.

These issues are addressed with the suboption `-Xshareclasses:enableBCI`, which is enabled by default. When using this suboption, any class modified by a JVMTI or java.lang.instrument agent is not stored in the cache. Classes which are not modified are stored as before. The `-Xshareclasses:enableBCI` suboption causes the
JVM to store original class byte data in the cache, which allows the ClassFileLoadHook event to be triggered for all classes loaded from the cache.

Using this option can improve the startup performance when JVMTI agents, java.lang.instrument agents, or both, are being used to modify classes. If you do not use this option, the JVM is forced to load classes from disk and find the equivalent class in the shared cache by doing a comparison. Because loading from disk and class comparison is done for every class loaded, the startup performance can be affected, as described in "Potential problems with runtime bytecode modification" on page 472. Using -Xshareclasses:enableBCI loads unmodified classes directly from the shared cache, improving startup performance, while still allowing these classes to be modified by the JVMTI agents, java.lang.instrument agents, or both.

Using -Xshareclasses:enableBCI with a modification context is still valid. However, -Xshareclasses:enableBCI prevents modified classes from being stored in the cache. Although unmodified classes are stored in the cache and logically separated by the specified modification context, using a modification context with -Xshareclasses:enableBCI does not provide any benefits and should be avoided.

JVMTI redefinition and retransformation of classes:

Redefined classes are never stored in the cache. Retransformed classes are not stored in the cache by default, but caching can be enabled using the -Xshareclasses:cacheRetransformed option.

Redefined classes are classes containing replacement bytecode provided by a JVMTI agent at run time, typically where classes are modified during a debugging session. Redefined classes are never stored in the cache.

Retransformed classes are classes with registered retransformation capable agents that have been called by a JVMTI agent at run time. Unlike RedefineClasses, the RetransformClasses function allows the class definition to be changed without reference to the original bytecode. An example of retransformation is a profiling agent that adds or removes profiling calls with each retransformation. Retransformed classes are not stored in the cache by default, but caching can be enabled using the -Xshareclasses:cacheRetransformed option. This option will also work with modification contexts or partitions.

The option -Xshareclasses:enableBCI is now enabled by default. However, if you use the -Xshareclasses:cacheRetransformed option, this option forces cache creation into -Xshareclasses:disableBCI mode. For more information see, "-Xshareclasses” on page 560.

Further considerations for runtime bytecode modification:

There are a number of additional items that you need to be aware of when using the cache with runtime bytecode modification.

If bytecode is modified by a non-JVMTI agent and defined using the JVM’s application class loader when shared classes are enabled, these modified classes are stored in the cache and nothing is stored to indicate that these are modified classes. Another JVM using the same cache will therefore load the classes with these modifications. If you are aware that your JVM is storing modified classes in the cache using a non-JVMTI agent, you are advised to use a modification context with that JVM to protect other JVMs from the modifications.
Combining partitions and modification contexts is possible but not recommended, because you will have partitions inside partitions. In other words, a partition A stored under modification context X will be different from partition A stored under modification context B.

Because the shared class cache is a fixed size, storing many different versions of the same class might require a much larger cache than the size that is typically required. However, note that the identical classes are never duplicated in the cache, even across modification contexts or partitions. Any number of metadata entries might describe the class and where it came from, but they all point to the same class bytes.

If an update is made to the file system and the cache marks a number of classes as stale as a result, note that it will mark all versions of each class as stale (when versions are stored under different modification contexts or partitions) regardless of the modification context being used by the JVM that caused the classes to be marked stale.

**Understanding dynamic updates**

The shared class cache must respond to file system updates; otherwise, a JVM might load classes from the cache that are out of date or “stale”. After a class has been marked stale, it is not returned by the cache if it is requested by a class loader. Instead, the class loader must reload the class from disk and store the updated version in the cache.

The cache is managed in a way that helps ensure that the following challenges are addressed:

- Java archive and compressed files are usually locked by class loaders when they are in use. The files can be updated when the JVM shuts down. Because the cache persists beyond the lifetime of any JVM using it, subsequent JVMs connecting to the cache check for Java archive and compressed file updates.
- `.class` files that are not in a `.jar` file can be updated at any time during the lifetime of a JVM. The cache checks for individual class file updates.
- `.class` files can be created or removed from directories found in classpaths at any time during the lifetime of a JVM. The cache checks the classpath for classes that have been created or removed.
- `.class` files must be in a directory structure that reflects their package structure. This structure helps ensure that when checking for updates, the correct directories are searched.

Class files contained in `.jar` files and compressed files, and class files stored as `.class` files on the file system, are accessed and used in different ways. The result is that the cache treats them as two different types. Updates are managed by writing file system time stamps into the cache.

Classes found or stored using a `SharedClassTokenHelper` cannot be maintained in this way, because Tokens are meaningless to the cache. As a direct consequence of updating the class data, AOT data is automatically updated.

**Storing classes**

When a classpath is stored in the cache, the Java archive and compressed files are time stamped. These time stamps are stored as part of the classpath. Directories are not time stamped. When a ROMClass is stored, if it came from a `.class` file on the file system, the `.class` file it came from is time stamped and this time stamp is
stored. Directories are not time stamped because there is no guarantee that
subsequent updates to a file cause an update to the directory holding the file.

If a compressed or Java archive file does not exist, the classpath containing it can
still be added to the cache, but ROMClasses from this entry are not stored. If an
attempt is made to add a ROMClass to the cache from a directory, but the
ROMClass does not exist as a .class file, it is not stored in the cache.

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

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

Finding classes

When the JVM finds a class in the cache, it must make more checks than when it
stores a class.

When a potential match has been found, if it is a .class file on the file system, the
time stamps of the .class file and the ROMClass stored in the cache are
compared. Regardless of the source of the ROMClass (.jar or .class file), every
Java archive and compressed file entry in the calling classpath, up to and including
the index at which the ROMClass was “found”, must be checked for updates by
obtaining the time stamps. Any update might mean that another version of the
class being returned had already been added earlier in the classpath.

Additionally, any classpath entries that are directories might contain .class files
that “shadow” the potential match that has been found. Class files might be
created or deleted in these directories at any point. Therefore, when the classpath
is walked and .jar files and compressed files are checked, directory entries are also
checked to see whether any .class files have been created unexpectedly. This
check involves building a string by using the classpath entry, the package names,
and the class name, and then looking for the class file. This procedure is expensive
if many directories are being used in class paths. Therefore, using .jar files gives
better shared classes performance.

Marking classes as stale

When an individual .class file is updated, only the class or classes stored from
that .class file are marked “stale”.

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

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

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

Classes in the cache that have been loaded from c, d, and a are marked stale. Making a single update to one .jar file might cause many classes in the cache to be marked stale. To avoid massive duplication as classes are updated, stale classes can be marked as not stale, or “redeemed”, if it is proved that they are not in fact stale.

**Redeeming stale classes**

Because classes are marked stale when a class path update occurs, many of the classes marked stale might not have updated. When a class loader stores a class, and in doing so effectively “updates” a stale class, you can “redeem” the stale class if you can prove that it has not in fact changed.

For example, assume that class X is stored in a cache after obtaining it from location c, where c is part of the classpath a;b;c;d. Suppose a is updated. The update means that a might now contain a new version of class X. For this example, assume a does not contain a new version of class X. The update marks all classes loaded from b, c, and d as stale. Next, another JVM must load class X. The JVM asks the cache for class X, but it is stale, so the cache does not return the class. Instead, the class loader fetches class X from disk and stores it in the cache, again using classpath a;b;c;d. The cache checks the loaded version of X against the stale version of X and, if it matches, the stale version is “redeemed”.

**AOT code**

A single piece of AOT code is associated with a specific method in a specific version of a class in the cache. If new classes are added to the cache as a result of a file system update, new AOT code can be generated for those classes. If a particular class becomes stale, the AOT code associated with that class also becomes stale. If a class is redeemed, the AOT code associated with that class is also redeemed. AOT code is not shared between multiple versions of the same class.

The total amount of AOT code can be limited using `-Xscmaxaot`, and cache space can be reserved for AOT code using `-Xscminaot`.

**JIT data**

JIT data is associated with a specific version of a class in the cache. If new classes are added to the cache as a result of a file system update, new JIT data can be generated for those classes. If a particular class becomes stale, the JIT data associated with that class also becomes stale. If a class is redeemed, the JIT data associated with that class is also redeemed. JIT data is not shared between multiple versions of the same class.

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

**Using the Java Helper API**

Classes are shared by the bootstrap class loader internally in the JVM. Any other Java class loader must use the Java Helper API to find and store classes in the shared class cache.
The Helper API provides a set of flexible Java interfaces so that Java class loaders to use the shared classes features in the JVM. The java.net.URLClassLoader shipped with the SDK has been modified to use a SharedClassURLClasspathHelper and any class loaders that extend java.net.URLClassLoader inherit this behavior. Custom class loaders that do not extend URLClassLoader but want to share classes must use the Java Helper API. This topic contains a summary on the different types of Helper API available and how to use them.

The Helper API classes are contained in the com.ibm.oti.shared package. For a detailed description of each helper and how to use it, see the Javadoc information.

**com.ibm.oti.shared.Shared**

The Shared class contains static utility methods: getSharedClassHelperFactory() and isSharingEnabled(). If \(-Xshareclasses\) is specified on the command line and sharing has been successfully initialized, isSharingEnabled() returns true. If sharing is enabled, getSharedClassHelperFactory() returns a com.ibm.oti.shared.SharedClassHelperFactory. The helper factories are singleton factories that manage the Helper APIs. To use the Helper APIs, you must get a Factory.

**com.ibm.oti.shared.SharedClassHelperFactory**

SharedClassHelperFactory provides an interface used to create various types of SharedClassHelper for class loaders. Class loaders and SharedClassHelpers have a one-to-one relationship. Any attempts to get a helper for a class loader that already has a different type of helper causes a HelperAlreadyDefinedException.

Because class loaders and SharedClassHelpers have a one-to-one relationship, calling findHelperForClassLoader() returns a Helper for a given class loader if one exists.

**com.ibm.oti.shared.SharedClassHelper**

There are three different types of SharedClassHelper:

- **SharedClassTokenHelper.** Use this Helper to store and find classes using a String token generated by the class loader. This Helper is normally used by class loaders that require total control over cache contents.
- **SharedClassURLHelper.** Store and find classes using a file system location represented as a URL. For use by class loaders that do not have the concept of a class path and load classes from multiple locations.
- **SharedClassURLClasspathHelper.** Store and find classes using a class path of URLs. For use by class loaders that load classes using a URL class path.

Compatibility between Helpers is as follows: Classes stored by SharedClassURLHelper can be found using a SharedClassURLClasspathHelper and the opposite also applies. However, classes stored using a SharedClassTokenHelper can be found only by using a SharedClassTokenHelper.

**Note:** Classes stored using the URL Helpers are updated dynamically by the cache (see “Understanding dynamic updates” on page 475). Classes stored by the SharedClassTokenHelper are not updated by the cache because the Tokens are meaningless Strings, so the Helper has no way of obtaining version information.

You can control the classes a URL Helper finds and stores in the cache using a SharedClassURLFilter. An object implementing this interface can be
passed to the SharedClassURLHelper when it is constructed or after it has been created. The filter is then used to decide which classes to find and store in the cache. See [Using the SharedClassHelper API](#) for more information. For a detailed description of each helper and how to use it, see the Javadoc information.

**Using the SharedClassHelper API:**

The SharedClassHelper API provides functions to find and store shared classes.

These functions are:

**findSharedClass**
Called after the class loader has asked its parent for a class, but before it has looked on disk for the class. If findSharedClass returns a class (as a byte[]), pass this class to defineClass(), which defines the class for that JVM and return it as a java.lang.Class object. The byte[] returned by findSharedClass is not the actual class bytes. The effect is that you cannot monitor or manipulate the bytes in the same way as class bytes loaded from a disk. If a class is not returned by findSharedClass, the class is loaded from disk (as in the nonshared case) and then the java.lang.Class defined is passed to storeSharedClass.

**storeSharedClass**
Called if the class loader has loaded class bytes from disk and has defined them using defineClass. Do not use storeSharedClass to try to store classes that were defined from bytes returned by findSharedClass.

**setSharingFilter**
Register a filter with the SharedClassHelper. The filter is used to decide which classes are found and stored in the cache. Only one filter can be registered with each SharedClassHelper.

You must resolve how to deal with metadata that cannot be stored. An example is when java.security.CodeSource or java.util.jar.Manifest objects are derived from .jar files. For each .jar file, the best way to deal with metadata that cannot be stored is always to load the first class from the .jar file. Load the class regardless of whether it exists in the cache or not. This load activity initializes the required metadata in the class loader, which can then be cached internally. When a class is then returned by findSharedClass, the function indicates where the class has been loaded from. The result is that the correct cached metadata for that class can be used.

It is not incorrect usage to use storeSharedClass to store classes that were loaded from disk, but which are already in the cache. The cache sees that the class is a duplicate of an existing class, it is not duplicated, and so the class continues to be shared. However, although it is handled correctly, a class loader that uses only storeSharedClass is less efficient than one that also makes appropriate use of findSharedClass.

**Filtering**

You can filter which classes are found and stored in the cache by registering an object implementing the SharedClassFilter interface with the SharedClassHelper. Before accessing the cache, the SharedClassHelper functions performs filtering using the registered SharedClassFilter object. For example, you can cache classes inside a particular package only by creating a suitable filter. To define a filter, implement the SharedClassFilter interface, which defines the following methods:
boolean acceptStore(String className)
boolean acceptFind(String className)

You must return true when you implement these functions so that a class can be found or stored in the cache. Use the supplied parameters as required. Make sure that you implement functions that do not take long to run because they are called for every find and store. Register a filter on a SharedClassHelper using the setSharingFilter(SharedClassFilter filter) function. See the Javadoc for the SharedClassFilter interface for more information.

**Applying a global filter**

You can apply a SharedClassFilter to all non-bootstrap class loaders that share classes. Specify the `com.ibm.oti.shared.SharedClassGlobalFilterClass` system property on the command line. For example:

```
-Dcom.ibm.oti.shared.SharedClassGlobalFilterClass=<filter class name>
```

**Obtaining information about shared caches:**

Use these APIs to obtain information about shared caches.

**com.ibm.oti.shared.SharedClassStatistics**

The SharedClassStatistics class provides static utilities that return the total cache size and the amount of free bytes in the active cache.

**com.ibm.oti.shared.SharedClassUtilities**

You can use these APIs to get information about shared class caches in a directory, and to remove specified shared class caches. The type of information available for each cache includes:

- The cache name.
- The cache size.
- The amount of free space in the cache.
- An indication of compatibility with the current JVM.
- Information about the type of cache; persistent or non-persistent.
- The last detach time.
- The Java version that created the cache.
- Whether the cache is for a 32-bit or 64-bit JVM.
- Whether the cache is corrupted.

**com.ibm.oti.shared.SharedClassCacheInfo**

This class is used by com.ibm.oti.shared.SharedClassUtilities to store information about a shared class cache and provides API methods to retrieve that information.

For information about the IBM JVMTI extensions for shared class caches, see "Finding shared class caches" on page 508, and "Removing a shared class cache" on page 510.

**Understanding shared classes diagnostic output**

When running in shared classes mode, a number of diagnostic tools can help you. The verbose options are used at run time to show cache activity and you can use the printStats and printAllStats utilities to analyze the contents of a shared class cache.

This section tells you how to interpret the output.
Verbose output:

The `verbose` suboption of `-Xshareclasses` gives the most concise and simple diagnostic output on cache usage.

See "-Xshareclasses" on page 560. Verbose output will typically look like this:

```
java -Xshareclasses:name=myCache,verbose -Xscmx10k HelloWorld
```

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.

Related reference:

"-Xshareclasses" on page 560

Enables class sharing. This option can take a number of suboptions, some of which are cache utilities.

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 verbose10 suboption of -Xshareclasses. With a cold cache, you see trace like this example:

```
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 0... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 3... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 17... Failed.
Storing class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 17... Succeeded.
```

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

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

```
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 0... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 3... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 17... Succeeded.
```

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

VerboseHelper output:

You can also obtain diagnostic data from the Java SharedClassHelper API using the `verboseHelper` suboption.

The output is divided into information messages and error messages:

- Information messages are prefixed with:
Info for SharedClassHelper id <n>: <message>

- Error messages are prefixed with:
  Error for SharedClassHelper id <n>: <message>

Use the Java Helper API to obtain this output; see “Using the Java Helper API” on page 477.

verboseAOT output:

VerboseAOT provides output when compiled AOT code is being found or stored in the cache.

When a cache is being populated, you might see the following message:
Storing AOT code for ROMMethod 0x523B95C0 in shared cache... Succeeded.

When a populated cache is being accessed, you might see the following message:
Finding AOT code for ROMMethod 0x524EAEB8 in shared cache... Succeeded.

AOT code is generated heuristically. You might not see any AOT code generated at all for a small application.

printStats utility:

The printStats utility prints summary information about the specified cache to the standard error output. You can optionally specify one or more types of cache content, such as AOT data or tokens, to see more detailed information about that type of content. To see detailed information about all the types of content in the cache, use the printAllStats utility instead.

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

The following output shows example results after running the printStats utility without a parameter, to generate summary data only:

Cache created with:
- Xnolinenumbers = false
  BCI Enabled = true

Cache contains only classes with line numbers

base address = 0x00007F44FC8EA000
end address = 0x00007F44FD8CE000
allocation pointer = 0x00007F44FCA1E650

cache size = 16776608
free bytes = 5638158
ROMClass bytes = 1263184
AOT bytes = 20164
Reserved space for AOT bytes = -1
Maximum space for AOT bytes = -1
JIT data bytes = 1120
Reserved space for JIT data bytes = -1
Maximum space for JIT data bytes = -1
Zip cache bytes = 1149512
Data bytes = 114080
Metadata bytes = 22348
Metadata % used = 0%
Class debug area size = 1331200
Class debug area used bytes = 179434
Class debug area % used = 13%

# ROMClasses = 433
# AOT Methods = 12
# Classpaths = 1
# URLs = 0
# Tokens = 0
# Zip caches = 21
# Stale classes = 0
% Stale classes = 0%

Cache is 66% full
Cache is accessible to current user = true

In the example output, -Xnolinenumbers = false means the cache was created without the -Xnolinenumbers option being specified.

BCI Enabled = true indicates that the cache was created with the -Xshareclasses:enableBCI suboption. This suboption is the default.

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 477.

# 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 473. For more information about modification contexts, see “Modification contexts” on page 472.

# Zip caches
The number of .zip files 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 475.

% 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 as follows:

\[
\text{% Full} = \left(\frac{('Cache Size' - 'Debug Area Size' - 'Free Bytes') \times 100}{('Cache Size' - 'Debug Area Size')}\right)
\]

Cache is accessible to current user
Whether the current user can access the cache.

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 486.

If you want to specify more than one type of cache content, use the plus symbol (+) to separate the values:

printStats=[type_1[+type_2][...]]

For example, use printStats=classpath to see a list of class paths that are stored in the shared cache, or printStats=romclass+url to see information about ROMClasses and URLs.

The following data types are valid. The values are not case sensitive:

- **Help** Prints a list of valid data types.
- **All** Prints information about all the following data types in the shared cache. This output is equivalent to the output produced by the printAllStats utility.
- **Classpath** Lists the class paths that are stored in the shared cache.
- **URL** Lists the URLs that are stored in the shared cache.
- **Token** Lists the tokens that are stored in the shared cache.
ROMClass
Prints information about the ROMClasses that are stored in the shared cache. This parameter does not print information about ROMMethods in ROMClasses.

ROMMethod
Prints ROMClasses and the ROMMethods in them.

AOT
Prints information about AOT compiled code in the shared cache.

JITprofile
Prints information about JIT data in the shared cache.

JIThint
Prints information about JIT data in the shared cache.

ZipCache
Prints information about zip entry caches that are stored in the shared cache.

printAllStats utility:
The printAllStats utility is a suboption of -Xshareclasses, optionally taking a cache name using name=<name>. This utility lists the cache contents in order, providing as much diagnostic information as possible. Because the output is listed in chronological order, you can interpret it as an “audit trail” of cache updates. Because it is a cache utility, the JVM displays the information about the cache specified or the default cache and then exits.

Each JVM that connects to the cache receives a unique ID. Each entry in the output is preceded by a number indicating the JVM that wrote the data.

Classpaths
1: 0x2234FA6C CLASSPATH
C:\myJVM\sdk\jre\lib\vm.jar
C:\myJVM\sdk\jre\lib\core.jar
C:\myJVM\sdk\jre\lib\charsets.jar
C:\myJVM\sdk\jre\lib\graphics.jar
C:\myJVM\sdk\jre\lib\security.jar
C:\myJVM\sdk\jre\lib\bnmpkcj.jar
C:\myJVM\sdk\jre\lib\bmqorb.jar
C:\myJVM\sdk\jre\lib\bmcjfw.jar
C:\myJVM\sdk\jre\lib\bmorbapi.jar
C:\myJVM\sdk\jre\lib\bmjcefw.jar
C:\myJVM\sdk\jre\lib\bjmgssprovider.jar
C:\myJVM\sdk\jre\lib\bmjssseprovider2.jar
C:\myJVM\sdk\jre\lib\bjmjaas1m.jar
C:\myJVM\sdk\jre\lib\bjmjaasactivelm.jar
C:\myJVM\sdk\jre\lib\bmcertxpathprovider.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 0x2234F70C, 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
```

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

**JIT data**

```
1: 0xD6290368 JITPROFILE: getKeyHash Signature: ()I Address: 0xD55118C0
   for ROMClass java/util/Hashtable$Entry at 0xD5511640.
2: 0xD6283848 JITHINT: loadClass Signature: (Ljava/lang/String;)Ljava/lang/Class; Address: 0xD5558F98
   for ROMClass com/ibm/oti/vm/BootstrapClassLoader at 0xD5558AE0.
```

The JIT stores data in the shared classes cache in the form of JITPROFILE and JITHINT entries to improve runtime performance. These outputs expose the content of the shared cache and can be useful for diagnostic purposes.

**Debugging problems with shared classes**

The following sections describe some of the situations you might encounter with shared classes and also the tools that are available to assist in diagnosing problems.

**Using shared classes trace:**

Use shared classes trace output only for debugging internal problems or for a detailed trace of activity in the shared classes code.
You enable shared classes trace using the `j9shr` trace component as a suboption of `-Xtrace`. See “Tracing Java applications and the JVM” on page 393 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 475, 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 471.

*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 471, 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 245.

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 245. 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 read/write access for the user only by default. Two users cannot share the same cache unless the -Xshareclasses:groupAccess command-line option is used when the cache is created. Service refresh 2 and later: The -Xshareclasses:groupAccess option must also be specified when the cache is accessed, unless the accessing user is also the cache creator. 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 HPROF Profiler**
HPROF is a demonstration profiler shipped with the IBM SDK that uses the JVMTI to collect and record information about Java execution. You can use HPROF to work out which parts of a program are using the most memory or processor time.

**Note:** For analyzing memory usage, you should use IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer, which is a newer tool. For more information about this tool, see “Using the IBM Monitoring and Diagnostic Tools” on page 319.

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
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 https://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html.

The command java -Xrunhprof:help shows the options available:

heap=dump|sites|all
This option helps in the analysis of memory usage. It tells HPROF to generate stack traces, from which you can see where memory was allocated. If you use the heap=dump option, you get a dump of all live objects in the heap. With heap=sites, you get a sorted list of sites with the most heavily allocated objects at the start. The default value all gives both types of output.

cpu=samples|times|old
The cpu option provides information that is useful in determining where the processor spends most of its time. If cpu is set to samples, the JVM pauses execution and identifies which method call is active. If the sampling rate is high enough, you get a good picture of where your program spends most of its time. If cpu is set to times, you receive precise measurements of how many times each method was called and how long each execution took. Although this option is more accurate, it slows down the program. If cpu is set to old, the profiling data is produced in the old HPROF format.

interval=y|n
The interval option applies only to cpu=samples and controls the time that the sampling thread sleeps between samples of the thread stacks.

monitor=y|n
The monitor option can help you understand how synchronization affects the performance of your application. Monitors implement thread synchronization. Getting information about monitors can tell you how much time different threads are spending when trying to access resources that are already locked. HPROF also gives you a snapshot of the monitors in use. This information is useful for detecting deadlocks.

format=a|b
The default for the output file is ASCII format. Set format to 'b' if you want to specify a binary format, which is required for some utilities like the Heap Analysis Tool.

file=<filename>
Use the file option to change the name of the output file. The default name for an ASCII file is java.hprof.txt. The default name for a binary file is java.hprof.

force=y|n
Typically, the default (force=y) overwrites any existing information in the output file. So, if you have multiple JVMs running with HPROF enabled, use force=n, which appends additional characters to the output file name as needed.

net=<host>:<port>
To send the output over the network rather than to a local file, use the net option.
**depth=<size>**

The depth option indicates the number of method frames to display in a stack trace. The default is 4.

**thread=y|n**

If you set the thread option to y, the thread id is printed next to each trace. This option is useful if you cannot see which thread is associated with which trace. This type of problem might occur in a multi-threaded application.

**doe=y|n**

The default behavior is to collect profile information when an application exits. To collect the profiling data during execution, set doe (dump on exit) to n.

**msa=y|n**

This feature is unsupported on IBM SDK platforms.

**cutoff=<value>**

Many sample entries are produced for a small percentage of the total execution time. By default, HPROF includes all execution paths that represent at least 0.0001 percent of the time spent by the processor. You can increase or decrease that cutoff point using this option. For example, to eliminate all entries that represent less than one-fourth of one percent of the total execution time, you specify cutoff=0.0025.

**verbose=y|n**

This option generates a message when dumps are taken. The default is y.

**lineno=y|n**

Each frame typically includes the line number that was processed, but you can use this option to suppress the line numbers from the output listing. If enabled, each frame contains the text Unknown line instead of the line number.

```
TRACE 1056:
  java/util/Locale.toUpperCase(Locale.java:Unknown line)
  java/util/Locale.<init>(Locale.java:Unknown line)
  java/util/Locale.<clinit>(Locale.java:Unknown line)
  sun/io/CharacterEncoding.aliasName(CharacterEncoding.java:Unknown line)
```

**Explanation of the HPROF output file**

The first section of the file contains general header information such as an explanation of the options, copyright, and disclaimers. A summary of each thread follows.

You can see the output after using HPROF with a simple program, shown as follows. This test program creates and runs two threads for a short time. From the output, you can see that the two threads called apples and then oranges were created after the system-generated main thread. Both threads end before the main thread. For each thread its address, identifier, name, and thread group name are displayed. You can see the order in which threads start and finish.

```
THREAD START (obj=11199050, id = 1, name="Signal dispatcher", group="system")
THREAD START (obj=111a2120, id = 2, name="Reference Handler", group="system")
THREAD START (obj=111ad910, id = 3, name="Finalizer", group="system")
THREAD START (obj=8b87a0, id = 4, name="main", group="main")
THREAD END (id = 4)
THREAD START (obj=11262d18, id = 5, name="Thread-0", group="main")
THREAD START (obj=112e9258, 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
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
classLoader 1128fa50
inheritedAccessControlContext 111aa2f0
threadLocals 111bea08
inheritedThreadLocals 111bea08
ARR 8bb978 (sz=4, trace=2, nelems=0, elem type=java/io/ObjectStreamField@8bac80)

If you set the heap option to sites or all, you get a list of each area of storage allocated by your code. The parameter all combines dump and sites. This list is ordered starting with the sites that allocate the most memory:

```
SITES BEGIN (ordered by live bytes) Tue Feb 06 10:54:46 2007
  rank  self  accum  live  allocated  stack  class
         % %      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  24  40308  24  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   300060 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.HashMap$Entry
 15  0.28% 55.57%   2640  15   2640  15   300000 char[]
 16  0.27% 55.84%   2476  17   2476  17   300000 java.util.HashMap$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 use this interface, including IBM Monitoring and Diagnostic Tools - Health Center. For more information about IBM Monitoring and Diagnostic Tools, see “Using the IBM Monitoring and Diagnostic Tools” on page 319.

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 [https://docs.oracle.com/javase/7/docs/technotes/guides/management/index.html](https://docs.oracle.com/javase/7/docs/technotes/guides/management/index.html).

For advice on porting JVMPI-based profilers to JVMTI, see [http://www.oracle.com/technetwork/articles/javase/jvmpitransition-138768.html](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](http://www.oracle.com/technetwork/articles/javase/jvmti-136367.html).

### IBM JVMTI extensions

The IBM SDK provides extensions to the JVMTI. The sample shows you how to write a simple JVMTI agent that uses these extensions.

The IBM SDK extensions to JVMTI allow a JVMTI agent to do the following tasks:
• Modify a dump.
• Modify a trace.
• Subscribe to and unsubscribe from JVM tracepoints
• Modify the logging configuration of the JVM.
• Start a JVM dump.
• Query the native memory use of the runtime environment.
• Find and remove shared class caches.
• Subscribe to and unsubscribe from verbose garbage collection logging.

The definitions that you need when you write a JVMTI agent are provided in the header files `jvmti.h` and `ibmjvmti.h`. These files are in `sdk/include`.

The sample JVMTI agent consists of two functions:
1. Agent_OnLoad()
2. DumpStartCallback()

**Agent_OnLoad()**

This function is called by the JVM when the agent is loaded at JVM startup, which allows the JVMTI agent to modify JVM behavior before initialization is complete. The sample agent obtains access to the JVMTI interface by using the JNI Invocation API function `GetEnv()`. The agent calls the APIs `GetExtensionEvents()` and `GetExtensionFunctions()` to find the JVMTI extensions that are supported by the JVM. These APIs provide access to the list of extensions available in the `jvmtiExtensionEventInfo` and `jvmtiExtensionFunctionInfo` structures. The sample uses an extension event and an extension function in the following way:

The sample JVMTI agent searches for the extension event `VmDumpStart` in the list of `jvmtiExtensionEventInfo` structures, by using the identifier `COM_IBM_VM_DUMP_START` provided in `ibmjvmti.h`. When the event is found, the JVMTI agent calls the JVMTI interface `SetExtensionEventCallback()` to enable the event, providing a function `DumpStartCallback()` that is called when the event is triggered.

Next, the sample JVMTI agent searches for the extension function `SetVMDump` in the list of `jvmtiExtensionFunctionInfo` structures, by using the identifier `COM_IBM_SET_VM_DUMP` provided in `ibmjvmti.h`. The JVMTI agent calls the function by using the `jvmtiExtensionFunction` pointer to set a JVM dump option `java:events=thrstart`. This option requests the JVM to trigger a Javadump every time a VM thread is started.

**DumpStartCallback()**

This callback function issues a message when the associated extension event is called. In the sample code, `DumpStartCallback()` is used when the `VmDumpStart` event is triggered.

**Compiling and running the sample JVMTI agent**

Use this command to build the sample JVMTI agent on Linux:
```
 gcc -I<SDK_path>/include -o libtiSample.so -shared tiSample.c
```

where `<SDK_path>` is the path to your SDK installation.
To run the sample JVMTI agent, use the command:

```bash
java -agentlib:tiSample -version
```

When the sample JVMTI agent loads, messages are generated. When the JVMTI agent initiates a Javadoc, the message JVMDUMP010 is issued.

**Sample JVMTI agent:**

A sample JVMTI agent, written in C/C++, using the IBM JVMTI extensions.

```c
/*
  * 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 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\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++ < extensionEventCount) {
        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++;
    }

    /* 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 322. 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:

```c
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 322.

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 ibmjit.h.

To trigger a JVM dump, use:

```c
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 325. 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 ibmjit.h.

To reset the JVM dump options to the values at JVM initialization, use:

```c
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.

```c
void JNICALL VMDumpStart(jvmtiEnv *jvmti_env, JNIEnv* jni_env, char* label, char* event, char* detail)
```

The event function provides the dump file name, the name of the JVMTI event, and the detail string from the dump event. The detail string provides additional information about the event that triggered the dump. This information is the same as the information detailed in the JVMDUMP039I message. For example:

```
```

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 detail string. The string might be empty.

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:

```c
void JNICALL VMDumpEnd(jvmtiEnv *jvmti_env, JNIEnv* jni_env, char* label, char* event, char* detail)
```

The event function provides the dump file name, the name of the JVMTI event, and the detail string from the dump event. The detail string provides additional information about the event that triggered the dump. This information is the same as the information detailed in the JVMDUMP039I message. For example:

```
```

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 com.ibm.VmDumpEnd.

**detail**: The dump event detail string. The string might be empty.

**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:

```c
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 399.

**Parameters**:

- `jvmti_env`: JVMTI environment pointer.
- `option`: Enter the JVM trace option string.

**Returns**:

- JVMI_ERROR_NONE: Success.
- JVMI_ERROR_NULL_POINTER: The `option` parameter is null.
- JVMI_ERROR_OUT_OF_MEMORY: There is insufficient system memory to process the request.
- JVMI_ERROR_INVALID_ENVIRONMENT: The `jvmti_env` parameter is invalid.
- JVMI_ERROR_WRONG_PHASE: The extension has been called outside the JVMTI live phase.
- JVMI_ERROR_ILLEGAL_ARGUMENT: The `option` parameter contains an invalid `-Xtrace` string.

**Subscribing to JVM tracepoints**:

A JVMTI agent can subscribe to JVM tracepoints by using the RegisterTracePointSubscriber() API.

The RegisterTracePointSubscriber() API has the JVMTI Extension function identifier com.ibm.RegisterTracePointSubscriber. The identifier is declared as macro COM_IBM_REGISTER_TRACEPOINT_SUBSCRIBER in the ibmjvmti.h file.

To register a subscription to JVM tracepoints, use the following syntax:

```c
jvmtiError RegisterTracePointSubscriber(jvmtiEnv* jvmti_env, char *description, jvmtiTraceSubscriber subscriber, jvmtiTraceAlarm alarm, void *userData, void **subscriptionID)
```

An ASCII character string that describes the subscriber must be passed in.
A pointer to user data can be supplied. This pointer is passed to the subscriber and alarm functions each time these functions are called. This pointer can be a null value.

A pointer to a subscription ID must be supplied. This pointer is returned by the RegisterTracePointSubscriber call if successful. The value must be supplied to a future call to the DeregisterTracePointSubscriber API, which is used to unsubscribe from the JVM tracepoint.

Parameters:
- `jvmti_env`: A pointer to the JVMTI environment.
- `description`: A string that describes your subscriber.
- `subscriber`: A function of type `jvmtiTraceSubscriber`.
- `alarm`: A function pointer of type `jvmtiTraceAlarm`.
- `user_data`: User data that is passed to the subscriber function.
- `subscription_id`: A pointer to a subscription identifier that is returned.

Returns:
- `JVMTI_ERROR_NONE`: Success.
- `JVMTI_ERROR_NULL_POINTER`: One of the supplied parameters is null.
- `JVMTI_ERROR_OUT_OF_MEMORY`: There is insufficient system memory to process the request.
- `JVMTI_ERROR_INVALID_ENVIRONMENT`: The `jvmti_env` parameter is not valid.
- `JVMTI_ERROR_WRONG_PHASE`: The extension has been called outside the JVMTI live phase.
- `JVMTI_ERROR_NOT_AVAILABLE`: JVM trace is not available.
- `JVMTI_ERROR_INTERNAL`: An internal error occurred.

The subscriber function type

The `jvmtiTraceSubscriber` function is defined as follows:

```c
typedef jvmtiError (*jvmtiTraceSubscriber)(jvmtiEnv *jvmti_env, void *record, jlong length, void *user_data);
```

The subscriber function must be of type `jvmtiTraceSubscriber`, which is declared in `ibmjvmti.h`. This function is called with each tracepoint record that is selected through the `-Xtrace:external` option. The tracepoint record that is supplied to the subscriber function is valid only for the duration of the function. If the subscriber wants to save the data, the data must be copied elsewhere. If the subscriber function returns an error, the alarm function is called, the subscription is disconnected, and no further tracepoints are sent to the subscriber.

Function parameters:
- `jvmti_env`: A pointer to the JVMTI environment.
- `record`: A UTF-8 string that contains a tracepoint record.
- `length`: The number of UTF-8 characters in the tracepoint record.
- `user_data`: User data that is supplied when the subscriber is registered.
The alarm function type

The jvmtiTraceAlarm function is defined as follows:

typedef jvmtiError (*jvmtiTraceAlarm)(jvmtiEnv *jvmti_env, void *subscription_id, void *user_data);

The alarm function must be of type jvmtiTraceAlarm, which is declared in 1bmjvmti.h. This function is called if the subscriber function returns an error.

Alarm function parameters:
- jvmti_env: A pointer to the JVMTI environment.
- subscription_id: The subscription identifier.
- user_data: User data that is supplied when the subscriber is registered.

Unsubscribing from JVM tracepoints:

The DeregisterTracePointSubscriber() API can be called by a JVMTI agent to unsubscribe from JVM tracepoints.

The DeregisterTracePointSubscriber() API has the JVMTI Extension function identifier com.ibm.DeregisterTracePointSubscriber. The identifier is declared as macro COM_IBM_DEREGISTER_TRACEPOINT_SUBSCRIBER in the 1bmjvmti.h file.

To unsubscribe from JVM tracepoints, use the following syntax:

jvmtiError DeregisterTracePointSubscriber(jvmtiEnv* jvmti_env, void *userData, void *subscription_id)

You must supply the subscription identifier that is returned by the call to the RegisterTracePointSubscriber() API. After the DeregisterTracePointSubscriber() API is called, no further calls are made to the subscriber function.

Parameters:
- jvmti_env: A pointer to the JVMTI environment.
- subscription_id: The subscription identifier.

Returns:
- JVMTI_ERROR_NONE: Success.
- JVMTI_ERROR_NULL_POINTER: The subscription_id parameter is null.
- JVMTI_ERROR_OUT_OF_MEMORY: There is insufficient system memory to process the request.
- JVMTI_ERROR_INVALID_ENVIRONMENT: The jvmti_env parameter is not valid.
- JVMTI_ERROR_WRONG_PHASE: The extension has been called outside the JVMTI live phase.

Querying Java runtime environment native memory categories:

You can query the total native memory consumption of the runtime environment for each memory category using the GetMemoryCategories() API.

The GetMemoryCategories() API has the JVMTI Extension Function identifier com.ibm.GetMemoryCategories. The identifier is declared as macro COM_IBM_GET_MEMORY_CATEGORIES in 1bmjvmti.h.

Native memory is memory requested from the operating system using library functions such as malloc() and mmap(). Runtime environment native memory use
is grouped under high-level memory categories, as described in the Javadump
section “Native memory (NATIVEMEMINFO)” on page 346. 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 549. 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 549.

**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:

```c
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.

```c
jint (JNI_CALL *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:

```c
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
} jvmtiSharedCacheInfo;
```

Chapter 10. Troubleshooting and support 509
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:

```c
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_FAILEDOLDER_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 \texttt{internalErrorCode} for more information about the failure.

Subscribing to verbose garbage collection logging:

You can subscribe to verbose Garbage Collection (GC) data logging through an IBM JVMTI extension.

The RegisterVerboseGCSubscriber() API has the JVMTI Extension function identifier \texttt{com.ibm.RegisterVerboseGCSubscriber}. The identifier is declared as macro \texttt{COM\_IBM\_REGISTER\_VERBOSEG\_SUBSCRIBER} in ibmjvmti.h.

To register a subscription to verbose GC data logging, use:

\begin{verbatim}
jvmtiError RegisterVerboseGCSubscriber(jvmtiEnv* jvmti_env, char *description, jvmtiVerboseGCSubscriber subscriber, jvmtiVerboseGCAlarm alarm, void *user_data, void **subscription_id)
\end{verbatim}

An ASCII character string describing the subscriber must be passed in.

An arbitrary pointer to user data must be supplied. This pointer is passed to the subscriber and alarm functions each time these functions are called. This pointer can be null.

A pointer to a subscription ID must be supplied. This pointer is returned by the RegisterVerboseGCSubscriber call if successful. The value must be supplied to a future call to DeregisterVerboseGCSubscriber.

Parameters:
- \texttt{jvmti_env}: A pointer to the JVMTI environment.
- \texttt{description}: A string that describes your subscriber.
subscriber: A function of type jvmtiVerboseGCSubscriber.
alarm: A function pointer of type jvmtiVerboseGCAAlarm.
user_data: User data that is passed to the subscriber function.
subscription_id: A pointer to a subscription identifier that is returned.

Returns:
JVMTI_ERROR_NONE: Success.
JVMTI_ERROR_NULL_POINTER: One of the supplied parameters is null.
JVMTI_ERROR_OUT_OF_MEMORY: There is insufficient system memory to process the request.
JVMTI_ERROR_INVALID_ENVIRONMENT: The jvmti_env parameter is not valid.
JVMTI_ERROR_WRONG_PHASE: The extension has been called outside the JVMTI live phase.
JVMTI_ERROR_NOT_AVAILABLE: GC verbose logging is not available.
JVMTI_ERROR_INTERNAL: An internal error has occurred.

The subscriber function type

The jvmtiVerboseGCSubscriber function is called with the following parameters:
typedef jvmtiError (*jvmtiVerboseGCSubscriber)(jvmtiEnv *jvmti_env, const char *record, jlong length, void *user_data);

The subscriber function must be of type jvmtiVerboseGCSubscriber, which is declared in ibmjvmti.h. This function is called with each record of verbose logging data produced by the JVM. The verbose logging record supplied to the subscriber function is valid only for the duration of the function. If the subscriber wants to save the data, the data must be copied elsewhere. If the subscriber function returns an error, the alarm function is called, and the subscription is de-registered.

Alarm function parameters:
jvmti_env: A pointer to the JVMTI environment.
record: An ascii string that contains a verbose log record.
length: The number of ascii characters in the verbose log record.
user_data: User data supplied when the subscriber is registered.

The alarm function type

The jvmtiVerboseGCAAlarm function is called with the following parameters:
typedef jvmtiError (*jvmtiVerboseGCAAlarm)(jvmtiEnv *jvmti_env, void *subscription_id, void *user_data);

The alarm function must be of type jvmtiVerboseGCAAlarm, which is declared in ibmjvmti.h. This function is called if the subscriber function returns an error.

Alarm function parameters:
jvmti_env: A pointer to the JVMTI environment.
user_data: User data supplied when the subscriber is registered.
subscription_id: The subscription identifier.
Unsubscribing from verbose garbage collection logging:

You can unsubscribe from verbose Garbage Collection (GC) data logging through an IBM JVMTI extension.

The DeregisterVerboseGCSubscriber() API has the JVMTI Extension Function identifier com.ibm.DeregisterVerboseGCSubscriber. The identifier is declared as macro COM_IBM_DEREGISTER_VERBOSEGC_SUBSCRIBER in ibmjvmti.h.

To unsubscribe from verbose GC data logging, use:

```plaintext
jvmtiError DeregisterVerboseGCSubscriber(jvmti_env, void *userData, void *subscription_id)
```

You must supply the subscription ID returned by the call to RegisterVerboseGCSubscriber. The previously registered subscriber function is no longer called with future verbose logging records.

Parameters:
- `jvmti_env`: A pointer to the JVMTI environment.
- `subscription_id`: The subscription identifier.

Returns:
- JVMTI_ERROR_NONE: Success.
- JVMTI_ERROR_NULL_POINTER: The `subscription_id` parameter is null.
- JVMTI_ERROR_OUT_OF_MEMORY: There is insufficient system memory to process the request.
- JVMTI_ERROR_INVALID_ENVIRONMENT: The `jvmti_env` parameter is not valid.
- JVMTI_ERROR_WRONG_PHASE: The extension has been called outside the JVMTI live phase.

Using the Diagnostic Tool Framework for Java

The Diagnostic Tool Framework for Java (DTFJ) is a Java application programming interface (API) from IBM used to support the building of Java diagnostic tools. DTFJ works with data from a system dump or a Javadump.

Note: The IBM Monitoring and Diagnostic Tools use the DTFJ interface, and should provide all the functionality that you need. However, you can use DTFJ to write your own diagnostic tools if required.

For analysis of core dumps from Linux and AIX platforms, copies of executable files and libraries are required along with the system dump. You must run the `jextract` utility provided in the SDK to collect these files, see “Using jextract” on page 382. You must run `jextract` with the same SDK level, on the system that produced the system dump. The `jextract` utility compresses the dump, executable files, and libraries into a single compressed file for use in subsequent problem diagnosis.

For Java 7 SDKs on Windows and z/OS platforms, you do not need to run the `jextract` utility. For Java 6 and Java 5.0 SDKs containing versions of the IBM J9 virtual machine before V2.6, you must still run the `jextract` utility for all platforms.

To work with a Javadump, no additional processing is required.

The DTFJ API helps diagnostic tools access the following information:
• Memory locations stored in the dump (System dumps only)
• Relationships between memory locations and Java internals (System dumps only)
• Java threads running in the JVM
• Native threads held in the dump (System dumps only)
• Java classes and their class loaders that were present
• Java objects that were present in the heap (System dumps only)
• Java monitors and the objects and threads they are associated with
• Details of the workstation on which the dump was produced (System dumps only)
• Details of the Java version that was being used
• The command line that launched the JVM

If your DTFJ application requests information that is not available in the Javadump, the API will return null or throw a DataUnavailable exception. You might need to adapt DTFJ applications written to process system dumps to make them work with Javadumps.

DTFJ is implemented in pure Java and tools written using DTFJ can be cross-platform. Therefore, you can analyze a dump taken from one workstation on another (remote and more convenient) machine. For example, a dump produced on an AIX PPC workstation can be analyzed on a Windows Thinkpad.

This chapter describes DTFJ in:
• “Using the DTFJ interface”
• “DTFJ example application” on page 518

API documentation for the DTFJ interface can be found here: API documentation

Using the DTFJ interface
To create applications that use DTFJ, you must use the DTFJ interface. Implementations of this interface have been written that work with system dumps from IBM SDK for Java versions 1.4.2 and later, and Javadumps from IBM SDK for Java 6 and later.

All DTFJ implementations support the same interface, but the DTFJ implementations supplied in Version 5.0 and later are different to the implementation supplied in Version 1.4.2. The DTFJ implementations have different factory class names that you must use. The DTFJ implementation supplied in Version 1.4.2 does not work with system dumps from Version 5 or later, and the DTFJ implementations supplied in Version 5 and later do not work with system dumps from Version 1.4.2.

Figure 6 on page 517 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.

```java
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 374). 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)
```
In 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.

```java
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 .....

        } catch (IOException e) { /* handle exception */ }
    }
}
```

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:
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 virtual machine 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 Java runtime 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 346.

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 514.

In this example, the program iterates through every available Java thread and checks whether it is equal to any of the available image threads. When they are found to be equal, the program displays the following message: Found a match.

The example demonstrates:

- How to iterate down through the class hierarchy.
- How to handle CorruptData objects from the iterators.
- The use of the .equals method for testing equality between objects.

```java
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( );
                    }
                }
            }
        }
    }
}
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("\ttJavaThread "+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 that allows the user to monitor and manage the behavior of Java applications.

**Note:** Because JConsole consumes significant system resources, Oracle recommend its use only in development environments for creating prototypes. Remote monitoring is also recommended to isolate the JConsole application from the platform being monitored. An alternative tool set for use in production environments is the IBM Monitoring and Diagnostic Tools. For more information, see [IBM Monitoring and Diagnostic Tools](#).
The tool is built on the java.lang.management API. JConsole connects to applications that run on the same workstation or on a remote workstation. The applications must be configured to allow access.

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 helps you 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 that you want to monitor must be started with command-line options that make it accessible to JConsole from other systems or other users. JConsole can attach to processes owned by the same user on the same system without these options. The simplest set of options for monitoring are shown in the following example:

   -Dcom.sun.management.jmxremote.port=<port number>
   -Dcom.sun.management.jmxremote.authenticate=false
   -Dcom.sun.management.jmxremote.ssl=false

   The value for <port number> must be a free port on your system. In this example, the authenticate and ssl options prevent password authentication and encryption by using Secure Sockets Layer (SSL). Using these options allows JConsole, or any other JMX agent, to connect to your Java application if it has access to the specified port. Use these non-secure options only in a development or testing environment. For more information about configuring security options, see [https://docs.oracle.com/javase/7/docs/technotes/guides/jmx/overview/connectors.html](https://docs.oracle.com/javase/7/docs/technotes/guides/jmx/overview/connectors.html).

2. Start JConsole by typing `jconsole` on the command line. 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` on the command line. Your path must contain the bin directory of the SDK.


3. Click connect. JConsole starts and displays the summary tab.

**Using JConsole to monitor a Java application**

The JConsole Summary tab shows key details about the connected JVM. 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 garbage collection 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” on page 523](#).

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 effect on 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>**.

JConsole uses the Attach API to connect to an application and can connect only to other IBM Java virtual machines. If you experience problems when using JConsole to monitor a remote application, the root cause might be the Attach API. To diagnose problems, see [“Attach API problem determination” on page 308](#).

**Known Limitations**

**Using the local process list**

The local process list does not work. Use `localhost:<port>` in the **Remote Process** text entry 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 [https://docs.oracle.com/javase/7/docs/technotes/guides/management/index.html](https://docs.oracle.com/javase/7/docs/technotes/guides/management/index.html).
**MBeans and MXBeans**

MBeans and MXBeans can be used to provide information about the state of the Java virtual machine (JVM). IBM provides additional MXBeans that extend the monitoring and management capabilities.

MXBeans are a generalized variant of MBeans. Because MXBeans are constructed by using only a pre-defined set of data types, MXBeans can be referenced and used more easily by applications such as JConsole.

Start JConsole by running the command `jconsole` from a command line. When you connect to a running JVM, you see an MBeans tab. This tab displays a navigation tree that contains the MBeans exported by the JVM. The list of available MBeans depends on the version of Java that you are using. The `java.lang.management` package includes MBean categories such as **Memory**, **OperatingSystem**, and **GarbageCollector**.

Clicking an MBean category in the navigation tree shows you all the related MBeans that are available. Clicking an individual MBean shows you the information that the MBean extracts from the JVM, separated into the following sections:

**Attributes**
Information about the current state. You can use some MBeans to change the JVM options. For example, in the **Memory** MBean, you might select the `Verbose` option to enable `VerboseGC` logging output.

**Operations**
Detailed information from the JVM. For example, in the **Threading** MBean, you see thread information that helps you to monitor deadlocked threads.

**Notifications**
Notifications that are supported by the MBean. Applications such as JConsole receive information from the MBean by subscribing to these notifications.

**Info**
Details about the available notifications.

**IBM MXBeans**

IBM provides further MXBeans to extend the monitoring and management capabilities:

**GarbageCollectorMXBean**
For monitoring garbage collection operations. You can obtain data about GC collection times, heap memory usage, number of compactions, and the amount of total freed memory.

**MemoryMXBean**
For monitoring memory usage, including data about maximum and minimum heap sizes, and shared class caches sizes.

**MemoryPoolMXBean**
For monitoring the usage of the memory pool, where supported.

**OperatingSystemMXBean**
For monitoring operating system settings such as physical and virtual memory size, processor capacity, and processor utilization.
GuestOSMXBean
For monitoring guest operating system statistics, as seen by the host hypervisor. This data includes memory and processor usage.

HypervisorMXBean
This MXBean provides information to determine whether the operating system is running in a virtualized environment, such as a virtual machine or LPAR. A method is also provided to determine the vendor of the hypervisor, if the operating system is running in a virtualized environment.

For more information about the standard platform MBeans, see the Oracle API documentation for the java.lang.management package at https://docs.oracle.com/javase/7/docs/api/java/lang/management/package-summary.html

For information about IBM MXBeans, see the IBM API documentation: API documentation
Chapter 11. Reference

This part of the product documentation contains reference information.

The sections are:
- “CORBA minor codes” on page 596
- “Environment variables” on page 599
- “Command-line options”
- “Default settings for the JVM” on page 603

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 526
- “System property command-line options” on page 527
- “JVM command-line options” on page 549
- “JIT and AOT command-line options” on page 573
- “Garbage Collector command-line options” on page 577

Specifying command-line options

Although the command line is the traditional way to specify command-line options, you can also pass options to the Java virtual machine (VM) by using a manifest file, options files, and environment variables.

The sequence of the Java options on the command line defines which options take precedence during startup. Rightmost options have precedence over leftmost options. In the following example, the `-Xjit` option takes precedence:
```
java -Xint -Xjit myClass
```

Use single or double quotation marks for command-line options only when explicitly directed to do so. Single and double quotation marks have different meanings on different platforms, operating systems, and shells. Do not use `'-X<option>'` or `"-X<option>"`. Instead, you must use `-X<option>`. For example, do not use `'-Xmx500m'` and `"-Xmx500m"`. Write this option as `-Xmx500m`.

At startup, the list of VM arguments is constructed in the following order, with the lowest precedence first:

1. Certain options are created automatically by the VM, which specify arguments such as search paths and version information. The VM automatically adds `-Xoptionsfile=<vm_dir>/options.default` at the beginning of the command line, where `<vm_dir>` is the path specified in “Conventions and terminology” on page 62. You can modify the options.default file to include any options that you want to specify for your application instead of entering these options on
the command line. For more information about the construction of the file, see "-Xoptionsfile" on page 558. This order change is introduced with the J9 VM V2.7. For customers using the default J9 VM V2.6, these arguments are constructed later, as indicated.

2. Environment variables that are described in “JVM environment settings” on page 600 are translated into command-line options. For example, the following environment variable adds the parameter -Xrs to the list of arguments:

   export IBM_NOSIGHANDLER=<non_null_string>

3. The IBM_JAVA_OPTIONS environment variable. You can set command-line options using this environment variable. The options that you specify with this environment variable are added to the command line when a JVM starts in that environment.

   The environment variable can contain multiple blank-delimited argument strings, but must not contain comments. For example:
   
   export IBM_JAVA_OPTIONS="-Dmysysprop1=tcpip -Dmysysprop2=wait -Xdisablejavadump"

   Note: The environment variable JAVA_TOOLS_OPTIONS is equivalent to IBM_JAVA_OPTIONS and is available for compatibility with JVMTI.

4. Options that are specified on the command line. For example:

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

   The Java launcher adds some automatically generated arguments to this list, such as the names of the main class.

   You can also use the -Xoptionsfile parameter to specify JVM options. This parameter can be used on the command line, or as part of the IBM_JAVA_OPTIONS environment variable. The contents of an option file are expanded in place during startup. For more information about the structure and contents of this type of file, see "-Xoptionsfile" on page 558.

   To troubleshoot startup problems, you can check which options are used by a JVM. Append the following command-line option, and inspect the Javadump file that is generated:

   -Xdump:java:events=vmstart

   Here is an extract from a Javadump file that shows the options that are used:

   ....
   2CIUSERARG -Xdump:java:file=/home/test_javacore.txt,events=vmstop
   2CIUSERARG -Dtest.cmdlineOption=1
   2CIUSERARG -XXallowvmshutdown:true
   2CIUSERARG -Xoptionsfile=test1.test_options_file
   ....

**General command-line options**

Use these options to print help on assert-related options, set the search path for application classes and resources, print a usage method, identify memory leaks inside the JVM, print the product version and continue, enable verbose output, and print the product version.

- *cp, -classpath <directories and compressed or .jar files separated by : (; on Windows )>*

   Sets the search path for application classes and resources. If -classpath and -cp are not used, and the CLASSPATH environment variable is not set, the user classpath is, by default, the current directory (.).

- *help, -?*

   Prints a usage message.
-fullversion
  Prints the build and version information for the JVM.

-showversion
  Prints product version and continues.

-verbose:<option>[,<option>...]  
  Enables verbose output. Separate multiple options using commas. These options are available:

  **class**
  Writes an entry to stderr for each class that is loaded.

  **dynload**
  Provides detailed information as each bootstrap class is loaded by the JVM:
  - The class name and package
  - For class files that were in a .jar file, the name and directory path of the .jar
  - Details of the size of the class and the time taken to load the class
  The data is written out to stderr. An example of the output on a Windows platform follows:
  ```
  <Loaded java/lang/String from C:\sdk\jre\lib\vm.jar>
  <Class size 17258; ROM size 21080; debug size 0>
  <Read time 27368 usec; Load time 782 usec; Translate time 927 usec>
  ```

  **gc**
  Provide verbose garbage collection information.

  **init**
  Writes information to stderr describing JVM initialization and termination.

  **jni**
  Writes information to stderr describing the JNI services called by the application and JVM.

  **sizes**
  Writes information to stderr describing the active memory usage settings.

  **stack**
  Writes information to stderr describing the Java and C stack usage for each thread.

-version
  Prints the full build and version information for the JVM.

### System property command-line options

Use the system property command-line options to set up your system.

- **-Dname==<value>**
  Sets a system property.

- **-Dcom.ibm.CORBA.AcceptTimeout**
  This system property controls the amount of time that a ServerSocket waits in a call to accept().

- **-Dcom.ibm.CORBA.AcceptTimeout=<ms>**
  Where `<ms>` is the wait time in milliseconds. The acceptable range is 0 - 5000. If this property is not set, a default value of 0 is used, which prevents a timeout. If the property is set but the value that is provided is not valid, a value of 5000 is used.
-Dcom.ibm.CORBA.AllowUserInterrupt
When this property is set to true, remote method calls can be interrupted with a Thread.interrupt() call.

-Dcom.ibm.CORBA.AllowUserInterrupt=true
When set, the thread that is waiting for the call to return is stopped. 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() calls that are received while the ORB waits for the call to finish.

-Dcom.ibm.CORBA.BufferSize
This system property sets the number of bytes of a General Inter-ORB Protocol (GIOP) message that is read from a socket on the first attempt.

-Dcom.ibm.CORBA.BufferSize=<bytes>
Where <bytes> is in the range is 0 - 2147483647. The default size is 2048. A larger buffer size increases the probability of reading the whole message in one attempt. Such an action might improve performance. The minimum size that can be used is 24 bytes.

-Dcom.ibm.CORBA.CommTrace
This system property turns on wire tracing for the Object Request Broker (ORB), which is also known as Comm tracing.

-Dcom.ibm.CORBA.CommTrace=true|false
When you set this option to true, every incoming and outgoing GIOP message is sent to the trace log. You can set this property independently from -Dcom.ibm.CORBA.Debug. Use this property if you want to look only at the flow of information, and you do not want to debug the internal information. The default value for this property is false.

Related reference:

“-Dcom.ibm.CORBA.Debug” on page 529
This system property enables debugging for the Object Request Broker (ORB) and includes tracing options that control how much information is recorded.

“-Dcom.ibm.CORBA.Debug.Output” on page 530
This system property redirects Object Request Broker (ORB) trace output to a file, which is known as a trace log.

-Dcom.ibm.CORBA.ConnectionMultiplicity
This system property sets the number of concurrent connections to the server endpoint (target host-port).

-Dcom.ibm.CORBA.ConnectionMultiplicity=<n>
Where <n> is in the range 0 - 2147483647. 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 default value is 1. The number of parallel connections can never exceed the number of requesting threads. The number of concurrent threads is therefore a sensible maximum limit for this property.

Read the following article for best practices on tuning this property:

https://www.ibm.com/support/docview.wss?uid=swg21669697
-Dcom.ibm.CORBA.ConnectTimeout
This system property controls the maximum amount of time that the ORB waits when a connection is opened to another ORB.

-Dcom.ibm.CORBA.AcceptTimeout=<s>
Where <s> is the wait time in seconds. The acceptable range is 0 - 300. By default, a timeout is not specified.

-Dcom.ibm.CORBA.Debug
This system property enables debugging for the Object Request Broker (ORB) and includes tracing options that control how much information is recorded.

-Dcom.ibm.CORBA.Debug=value
Where value is one of the following options:
false  No output is produced. This option is the default value.
true   Messages and traces for the entire ORB code flow

Note: If you use this property without specifying a value, tracing is enabled.

If you want to limit tracing to specific subcomponents of ORB, you can also specify the -Dcom.ibm.CORBA.Debug.Component system property.

Related reference:

- “-Dcom.ibm.CORBA.Debug.Component”
  This system property can be used with -Dcom.ibm.CORBA.Debug=true to generate trace output for specific Object Request Broker (ORB) subcomponents such as MARSHAL or DISPATCH. This finer level of tracing helps you debug problems with ORB operations.

- “-Dcom.ibm.CORBA.Debug.Output” on page 530
  This system property redirects Object Request Broker (ORB) trace output to a file, which is known as a trace log.

- “-Dcom.ibm.CORBA.CommTrace” on page 528
  This system property turns on wire tracing for the Object Request Broker (ORB), which is also known as Comm tracing.

-Dcom.ibm.CORBA.Debug.Component
This system property can be used with -Dcom.ibm.CORBA.Debug=true to generate trace output for specific Object Request Broker (ORB) subcomponents such as MARSHAL or DISPATCH. This finer level of tracing helps you debug problems with ORB operations.

-Dcom.ibm.CORBA.Debug.Component=name
Where name can be one of the following ORB subcomponents:
  • DISPATCH
  • MARSHAL
  • TRANSPORT
  • CLASSLOADER
  • ALL

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

Note: This option has no effect unless it is used with the system property -Dcom.ibm.CORBA.Debug=true.
The following setting enables tracing for the DISPATCH, TRANSPORT, and CLASSLOADER components:

-Dcom.ibm.CORBA.Debug=true -Dcom.ibm.CORBA.Debug.Component=DISPATCH,TRANSPORT,CLASSLOADER

**Related reference:**

"-Dcom.ibm.CORBA.Debug" on page 529

This system property enables debugging for the Object Request Broker (ORB) and includes tracing options that control how much information is recorded.

"-Dcom.ibm.CORBA.Debug.Output" on page 529

This system property redirects Object Request Broker (ORB) trace output to a file, which is known as a trace log.

"-Dcom.ibm.CORBA.CommTrace” on page 528

This system property turns on wire tracing for the Object Request Broker (ORB), which is also known as Comm tracing.

**-Dcom.ibm.CORBA.Debug.Output**

This system property redirects Object Request Broker (ORB) trace output to a file, which is known as a trace log.

-Dcom.ibm.CORBA.Debug.Output=filename

Where *filename* is the name you want to specify for your trace log. If this property is not specified or the value of *filename* is empty, the file name defaults to the following format:

`orbtrace.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.

**Related reference:**

"-Dcom.ibm.CORBA.Debug” on page 529

This system property enables debugging for the Object Request Broker (ORB) and includes tracing options that control how much information is recorded.

"-Dcom.ibm.CORBA.CommTrace” on page 528

This system property turns on wire tracing for the Object Request Broker (ORB), which is also known as Comm tracing.

**-Dcom.ibm.CORBA.enableLocateRequest**

Setting this system property causes the ORB to send a LocateRequest before the actual Request.

-Dcom.ibm.CORBA.enableLocateRequest=[true|false]

The default for this option is false.

**-Dcom.ibm.CORBA.FragmentSize**

This system property controls General Inter-ORB Protocol (GIOP) 1.2 fragmentation.

-Dcom.ibm.CORBA.FragmentSize=<bytes>

Where *<bytes>* is in the range is 0 - 2147483647. The default size is 1024 bytes
in the initial release, and 4096 bytes in service refresh 1 and later. The size that
is specified is rounded down to the nearest multiple of 8, with a minimum size
of 64 bytes. To turn off message fragmentation, set the value to 0.

-Dcom.ibm.CORBA.FragmentTimeout
This system property controls the maximum length of time for which the ORB
waits for second and subsequent message fragments before the ORB times out.

-Dcom.ibm.CORBA.FragmentTimeout=<ms>
Where <ms> is milliseconds in the range 0 - 600000. The default time is 300000
milliseconds. Set the value to 0 if a timeout is not required.

-Dcom.ibm.CORBA.GIOPAddressingDisposition
This system property sets the addressing disposition for a General Inter-ORB
Protocol (GIOP) 1.2 Request, LocateRequest, Reply, or LocateReply.

-Dcom.ibm.CORBA.GIOPAddressingDisposition=[0|1|2]
Where the following options apply:
• 0 = Object Key
• 1 = GIOP Profile
• 2 = Full IOR

If this property is not set or is passed a value that is not valid, a default of 0 is
used.

-Dcom.ibm.CORBA.InitialReferencesURL
Use this property if you do not have a bootstrap server and want to have a file on
the web server that serves the purpose.

-Dcom.ibm.CORBA.InitialReferencesURL=<url>
Where <url> is a correctly formed URL, for example, http://w3.mycorp.com/
InitRefs.file. The InitRefs.file file contains a name and value pair such as
NameService=<stringified_IOR>. If you specify this property, the ORB does not
attempt the bootstrap approach.

Note: This property is deprecated.

-Dcom.ibm.CORBA.ListenerPort
This system property sets the port on which the server listens for incoming
requests.

-Dcom.ibm.CORBA.ListenerPort=<port>
Where <port> is in the range is 0 - 2147483647. By default, the next available
port is selected. If this property is specified, the ORB starts to listen during
ORB initialization.

-Dcom.ibm.CORBA.LocalHost
This system property sets the host name or IP address of the system on which the
ORB is running.

-Dcom.ibm.CORBA.LocalHost=<host>
Where <host> is an IP address or a host name. If this property is not set,
retrieve the local host by calling InetAddress.getLocalHost().getHostAddress().
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.

-Dcom.ibm.CORBA.LocateRequestTimeout
This system property defines the number of seconds to wait before the ORB times
out on a LocateRequest message.
-Dcom.ibm.CORBA.LocateRequestTimeout=<s>
    Where <s> is seconds in the range 0 - 2147483647. The default time is 0, which prevents timing out.

-Dcom.ibm.CORBA.MaxOpenConnections
This system property determines the maximum number of in-use connections that are to be kept in the connection cache table at any one time.

-Dcom.ibm.CORBA.MaxOpenConnections=<number>
    Where <number> is the number of connections in the range 0 - 2147483647. The default value is 240.

-Dcom.ibm.CORBA.MinOpenConnections
This system property determines the desired minimum number of in-use connections that are to be kept in the connection cache table at any one time.

-Dcom.ibm.CORBA.MinOpenConnections=<number>
    Where <number> is the number of connections in the range 0 - 2147483647. The default value is 100.

    Note: The ORB cleans up only connections that are not busy from the connection cache table, if the size is of the table is larger than <number>.

-Dcom.ibm.CORBA.NoLocalInterceptors
This system property can be used to prevent the use of local portable interceptors.

-Dcom.ibm.CORBA.NoLocalInterceptors=[true|false]
    The default value is false. The expected result is improved performance if interceptors are not required when the ORB connects to a co-located object.

-Dcom.ibm.CORBA.ORBCharEncoding
This system property sets the native encoding set for the ORB.

-Dcom.ibm.CORBA.ORBCharEncoding=<encoding>
    Where <encoding> is a native encoding set. The default value is ISO8859_1.

-Dcom.ibm.CORBA.ORBWCharDefault
This system property indicates the wchar code set that should be used with other ORBs that do not publish a wchar code set.

-Dcom.ibm.CORBA.ORBWCharDefault=<code_set>
    Where <code_set> is a wchar code set. The default value is UCS2.

-Dcom.ibm.CORBA.requestRetriesCount
This system property governs the number of request retries that are attempted.

-Dcom.ibm.CORBA.requestRetriesCount=<value>
    Where <value> is in the range 1 - 10. The default value is 1.

    Use this property with the -Dcom.ibm.CORBA.requestRetriesDelay system property to control ORB retry behavior.

Related reference:

"-Dcom.ibm.CORBA.requestRetriesDelay"
This property determines the time delay in milliseconds (ms) between two retries.

-Dcom.ibm.CORBA.requestRetriesDelay
This property determines the time delay in milliseconds (ms) between two retries.

-Dcom.ibm.CORBA.requestRetriesDelay=<ms>
    Where <ms> is in the range 0 - 60000 milliseconds. The default value is 0.
Use this property with the `-Dcom.ibm.CORBA.requestRetriesCount` system property to control ORB retry behavior.

Related reference:

```
-Dcom.ibm.CORBA.requestRetriesCount` on page 532
```

This system property governs the number of request retries that are attempted.

**-Dcom.ibm.CORBA.RequestTimeout**

This system property defines the number of seconds to wait before the ORB times out on a Request message.

```
-Dcom.ibm.CORBA.RequestTimeout=<s>
```

Where `<s>` is seconds in the range 0 - 2147483647. The default value is 0, which prevents timing out.

**-Dcom.ibm.CORBA.SendingContextRunTimeSupported**

This system property determines whether the CodeBaseSendingContextRunTime service is operating.

```
-Dcom.ibm.CORBA.SendingContextRunTimeSupported=[true|false]
```

The default value is true. When set to false, the ORB does not attach a SendingContextRuntime service context to outgoing messages.

**-Dcom.ibm.CORBA.SendVersionIdentifier**

This system property causes the ORB to send an initial dummy request in order to determine, from the response, the partner version of the remote ORB server.

```
-Dcom.ibm.CORBA.SendVersionIdentifier=[true|false]
```

The default value is false.

**-Dcom.ibm.CORBA.ServerSocketQueueDepth**

This system property controls the maximum queue length for incoming connection requests.

```
-Dcom.ibm.CORBA.ServerSocketQueueDepth=<length>
```

Where `<length>` is in the range is 50 - 2147483647. If a connection request arrives when the queue is full, the connection is refused. If the property is not set, a default length of 0 is used. If the property is not valid, 50 is used.

**-Dcom.ibm.CORBA.ShortExceptionDetails**

This system property controls the method by which information is returned following a CORBA system exception.

```
-Dcom.ibm.CORBA.ShortExceptionDetails=[false | true]
```

When a CORBA SystemException reply is created, the ORB includes, by default, 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. The default value for this property is false.

**-Dcom.ibm.dbgmalloc**

This option provides memory allocation diagnostic information for class library native code.

```
-Dcom.ibm.dbgmalloc=true
```

When an application is started with this option, a javadump records the amount of memory allocated by the class library components. You can use this option together with the `-Xcheck:memory` option to obtain information about class library call sites and their allocation sizes. Enabling this option has an
impact on throughput performance. For sample javadump output, see “Native memory (NATIVEMEMINFO)” on page 346.

-Dcom.ibm.enableClassCaching
Setting this property to true enables caching of the Latest User Defined Class Loader (LUDCL).

-Dcom.ibm.enableClassCaching=[true|false]
By reducing repeated lookups, Java applications that use deserialization extensively can see a performance improvement. The default value for this property is false.

-Dcom.ibm.enableLegacyDumpSecurity
To improve security, the security checks in certain com.ibm.jvm.Dump APIs are now enabled by default, when the SecurityManager is enabled. Use this system property to turn off security checking for these APIs.

Security checking is enabled in the following APIs:
- com.ibm.jvm.Dump.JavaDump()
- com.ibm.jvm.Dump.SnapDump()

-Dcom.ibm.enableLegacyDumpSecurity=false
This option turns off security checking.

To turn security checking back on, set -Dcom.ibm.enableLegacyDumpSecurity=true. This option is the default.

Related reference:
“-Dcom.ibm.enableLegacyTraceSecurity” on page 535
To improve security, the security checks in certain com.ibm.jvm.Trace APIs are now enabled by default, when the SecurityManager is enabled. Use this system property to turn off security checking for these APIs.

“-Dcom.ibm.enableLegacyLogSecurity”
To improve security, the security checks in certain com.ibm.jvm.Log APIs are now enabled by default, when the SecurityManager is enabled. Use this system property to turn off security checking for these APIs.

-Dcom.ibm.enableLegacyLogSecurity
To improve security, the security checks in certain com.ibm.jvm.Log APIs are now enabled by default, when the SecurityManager is enabled. Use this system property to turn off security checking for these APIs.

Security checking is enabled in the following APIs:
- com.ibm.jvm.Log.QueryOptions()
- com.ibm.jvm.Log.SetOptions(String)

-Dcom.ibm.enableLegacyLogSecurity=false
This option turns off security checking.

To turn security checking back on, set -Dcom.ibm.enableLegacyLogSecurity=true. This option is the default.

Related reference:
“-Dcom.ibm.enableLegacyDumpSecurity”
To improve security, the security checks in certain com.ibm.jvm.Dump APIs are now enabled by default, when the SecurityManager is enabled. Use this system property
property to turn off security checking for these APIs.

-Dcom.ibm.enableLegacyTraceSecurity

To improve security, the security checks in certain com.ibm.jvm.Trace APIs are now enabled by default, when the SecurityManager is enabled. Use this system property to turn off security checking for these APIs.

-Dcom.ibm.enableLegacyTraceSecurity=false

This option turns off security checking.

To turn security checking back on, set -Dcom.ibm.enableLegacyTraceSecurity=true. This option is the default.

Related reference:

-Dcom.ibm.enableLegacyDumpSecurity" on page 534

To improve security, the security checks in certain com.ibm.jvm.Dump APIs are now enabled by default, when the SecurityManager is enabled. Use this system property to turn off security checking for these APIs.

-Dcom.ibm.enableLegacyLogSecurity” on page 534

To improve security, the security checks in certain com.ibm.jvm.Log APIs are now enabled by default, when the SecurityManager is enabled. Use this system property to turn off security checking for these APIs.

-Dcom.ibm.lang.management.verbose

Enables verbose information from java.lang.management operations to be written to the output channel during VM operation.

-Dcom.ibm.lang.management.verbose

There are no options for this system property.

-Dcom.ibm.IgnoreMalformedInput

Invalid UTF8 or malformed byte sequences are replaced with the standard unicode replacement character \ufffd.

-Dcom.ibm.IgnoreMalformedInput=true

To retain the old behavior, where invalid UTF8 or malformed byte sequences are ignored, set this system property to true.

-Dcom.ibm.streams.CloseFDWithStream

Determines whether the close() method of a stream object closes a native file descriptor even if the descriptor is still in use by another stream object.
-Dcom.ibm.streams.CloseFDWithStream=[true | false]

Usually, you create a FileInputStream or FileOutputStream instance by passing a String or a File object to the stream constructor method. Each stream then has a separate file descriptor. However, you can also create a stream by using an existing FileDescriptor instance, for example one that you obtain from a RandomAccessFile instance, or another FileInputStream or FileOutputStream instance. Multiple streams can then share the same file descriptor.

If you set this option to false, when you use the close() method of the stream, the associated file descriptor is also closed only if it is not in use by any other streams. If you set the option to true, the file descriptor is closed regardless of any other streams that might still be using it.

The default setting is true.

Note: Before version 7 service refresh 5, the default behavior was to close the file descriptor only when all the streams that were using it were also closed. This system property exists so that you can revert to this previous default behavior if necessary. This system property will be removed in a future release, so you should adjust your applications to use the new default behavior before you upgrade to a later release.

-Dcom.ibm.tools.attach.enable
Enable the Attach API for this application.

-Dcom.ibm.tools.attach.enable=yes
The Attach API allows your application to connect to a virtual machine. Your application can then load an agent application into the virtual machine. The agent can be used to perform tasks such as monitoring the virtual machine status.

Related concepts:
“Support for the Java Attach API” on page 195

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.

-Dcom.ibm.tools.rmic.iiop.Debug
This system property causes the rmic compiler to report the mappings of fully qualified class names to short names, which is useful for debugging purposes.

-Dcom.ibm.tools.rmic.iiop.Debug=true | false
The rmic compiler 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. The default value is false.

-Dcom.ibm.tools.rmic.iiop.SkipImports
This system property causes the rmic compiler to generate classes only with fully qualified names.

-Dcom.ibm.tools.rmic.iiop.SkipImports=true | false
The default value is false.

-Dcom.ibm.xtq.processor.overrideSecureProcessing
This system property affects the XSLT processing of extension functions or extension elements when Java security is enabled.
Purpose

The use of extension functions or extension elements is no longer allowed when Java security is enabled. The system property can be used to revert to the behavior in earlier releases.

Parameters

-com.ibm.xtq.processor.overrideSecureProcessing=true
To revert to the behavior in earlier releases of the IBM SDK, set this system property to true.

-Dcom.ibm.zipfile.closeinputstreams
The Java.util.zip.ZipFile class allows you to create InputStreams on files held in a compressed archive.

-Dcom.ibm.zipfile.closeinputstreams=true
Under some conditions, using ZipFile.close() to close all InputStreams that have been opened on the compressed archive might result in a 56-byte-per-InputStream native memory leak. Setting the -Dcom.ibm.zipfile.closeinputstreams=true forces the JVM to track and close InputStreams without the memory impact caused by retaining native-backed objects. Native-backed objects are objects that are stored in native memory, rather than the Java heap. By default, the value of this system property is not enabled.

-Dfile.encoding
Use this property to define the file encoding that is required.

-Dfile.encoding=value
Where value defines the file encoding that is required.

By default the IBM GBK converter follows Unicode 3.0 standards. To force the IBM GBK converter to follow Unicode 2.0 standards, use a value of bestfit936.

-Dibm.disableAltProcessor
This option stops the ALT-key, when pressed, from highlighting the first menu in the active window of the user interface.

-Dibm.disableAltProcessor=true
Set this property on the command line to prevent the ALT-key from highlighting the first menu in the active window.

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

-Dibm.jvm.bootclasspath
The value of this property is used as an additional search path.

-Dibm.jvm.bootclasspath
The value of this property is used as an additional search path, which is inserted between any value that is defined by -Xbootclasspath/p: and the bootclass path. The bootclass path is either the default or the one that you defined by using the -Xbootclasspath: option.

-Dibm.stream.nio
This option addresses the ordering of IO and NIO converters.
-Dibm.stream.nio=[true | false]
   When this option is set to true, the NIO converters are used instead of the IO converters. By default the IO converters are used.

-Dil8n.vs
This system property enables awareness of Unicode Ideographic Variation Sequence (IVS) to draw characters, except in peered components.

-Dil8n.vs=[true]
The behavior depends on the font specified. If the font supports IVS, and has a glyph based on the combination of a base character and a variation selector character, an accurate glyph can be picked up. If not, the base character is displayed and the variation selector character is ignored. Because this option changes the behavior of the font drawing engine, the option is disabled by default. When disabled, a variation selector is displayed as an undefined character. This option is supported only for Japanese.

-DJAVABIDI
This system property specifies whether bidirectional layout transformations are run.

JAVABIDI=[S(<TOSHNALEYZ>), [U(<TOSHNALEYZ>), ]C(<codepage1;codepage2;...>)]

The default value for the JAVABIDI system property is NO, which specifies that no bidirectional layout transformations are run.

When the JAVABIDI system property is not set to NO, its value can contain 1 - 3 parts. Each part starts with a letter identifier followed by a set of values within parentheses.

The letter identifiers are as follows:
- S: the single-byte character set (SBCS) part, which describes the bidirectional attributes of the SBCS data that is consumed or produced by the conversions. This part designates the data as it is stored outside the Java runtime environment.
- U: the Unicode part, which describes the bidirectional attributes of the Unicode data that is consumed or produced by the conversions.
- C: the CodePage part, which specifies one or more encodings. If you specify this part, only data with encodings that are listed in this part are submitted to the bidirectional layout transformation. If you omit this part, the layout transformations are run for all encodings except Cp850.

See the subtopics for more information about the valid values for each part.

Note: Applications should not try to modify the value of the JAVABIDI property after the initialization of the Java virtual machine (VM). For performance reasons, VM implementations might check the value of the JAVABIDI property only at start-up time, so later changes have no effect.

Examples
JAVABIDI=U(1LYNNUNNNK), S(VLNSUNNNK), C(Cp420)
JAVABIDI=C(Cp420), S(VLNSUNNNK), U(1LYNNUNNNK)

The order of the part specifications is not significant.
JAVABIDI=U(1LYNNUNNNK), S(VDNS--NK), C(Cp420;IBM-420)

The hyphens in the S part represent default values for the corresponding symbols.
JAVABIDI=C(Cp420)
Because both the $S$ and the $U$ parts are omitted, those parts receive default values for all the symbols.

**Related concepts:**

"Support for bidirectional data" on page 227

The runtime environment includes support for bidirectional data, where text is written from right to left and also from left to right.

**S and U Parts:**

The $S$ and $U$ parts of the **JAVABIDI** system property describe the bidirectional attributes of the data that is consumed or produced by the bidirectional layout conversions. The $S$ part applies to single-byte character set (SBCS) data, and the $U$ part applies to Unicode data.

Each part has the following format:

```
part_id(TOSHNALEYZ)
```

Where $part_id$ is the part identifier, either $S$ or $U$. **TOSHNALEYZ** is a list of symbols, each of which is replaced in the command by a value, as described in the following table.

**Notes:**

1. The part identifier and the values are case-sensitive.
2. You can use a hyphen ("-") in place of a symbol to use the default value for that symbol.

**Table 21. Symbols for the $S$ and $U$ parts, their possible values and definitions**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Valid values and their meaning</th>
<th>Default value</th>
<th>Language applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Text type</td>
<td>I (implicit) V (visual)</td>
<td>V (for $S$ part) I (for $U$ part)</td>
<td>Arabic and Hebrew</td>
</tr>
<tr>
<td>O</td>
<td>Orientation</td>
<td>L (left to right) R (right to left) C (contextual left to right) D (contextual right to left)</td>
<td>L</td>
<td>Arabic and Hebrew</td>
</tr>
<tr>
<td>S</td>
<td>Swapping</td>
<td>Y (yes) N (no)</td>
<td>N (for $S$ part) Y (for $U$ part)</td>
<td>Arabic and Hebrew</td>
</tr>
<tr>
<td>H</td>
<td>Text shaping</td>
<td>N (nominal) S (shaped) I (initial) M (middle) F (final) B (isolated)</td>
<td>S (for $S$ part) N (for $U$ part)</td>
<td>Arabic only</td>
</tr>
<tr>
<td>N</td>
<td>Numerals</td>
<td>N (nominal) H (national) C (contextual)</td>
<td>N</td>
<td>Arabic only</td>
</tr>
<tr>
<td>A</td>
<td>Bidirectional algorithm</td>
<td>U (Unicode) R (roundtrip)</td>
<td>U</td>
<td>Arabic and Hebrew</td>
</tr>
<tr>
<td>L</td>
<td>Lam-Alef shaping option</td>
<td>N (Near) B (At Begin) E (At End) A (Auto)</td>
<td>A</td>
<td>Arabic only</td>
</tr>
</tbody>
</table>
Table 21. Symbols for the S and U parts, their possible values and definitions (continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Valid values and their meaning</th>
<th>Default value</th>
<th>Language applicability</th>
</tr>
</thead>
</table>
| E      | Seen Tail shaping option | B (At Begin)  
A (Auto)  
N (Near)  
E (At End) | A | Arabic only |
| Y      | Yeh-Hamza shaping option | E (At End)  
A (Auto)  
N (Near)  
O (One cell)  
B (At Begin) | A | Arabic only |
| Z      | Tashkeel shaping option | A (Auto)  
B (At Begin)  
E (At End)  
W (With Width)  
Z (Zero Width)  
K (Keep) | A | Arabic only |

C Part:

The C part of the JAVABIDI system property specifies one or more encodings to be operated on by the bidirectional layout conversions.

This part has the following format:

\[C(codepage1;codepage2;\ldots)\]

The variables `codepage1` and `codepage2` are one of the following bidirectional code pages.

Table 22. Supported bidirectional code pages

<table>
<thead>
<tr>
<th>Code page</th>
<th>Canonical name for NIO</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cp420</td>
<td>IBM-420</td>
<td>Arabic</td>
</tr>
<tr>
<td>Cp424</td>
<td>IBM-424</td>
<td>Hebrew</td>
</tr>
<tr>
<td>Cp856</td>
<td>IBM-856</td>
<td>Hebrew</td>
</tr>
<tr>
<td>Cp862</td>
<td>IBM-862</td>
<td>Hebrew</td>
</tr>
<tr>
<td>Cp864</td>
<td>IBM-864</td>
<td>Arabic</td>
</tr>
<tr>
<td>Cp867</td>
<td>IBM-867</td>
<td>Hebrew</td>
</tr>
<tr>
<td>Cp1046</td>
<td>IBM-1046</td>
<td>Arabic</td>
</tr>
<tr>
<td>Cp1255</td>
<td>windows-1255</td>
<td>Hebrew</td>
</tr>
<tr>
<td>Cp1256</td>
<td>windows-1256</td>
<td>Arabic</td>
</tr>
<tr>
<td>ISO8859_6</td>
<td>ISO8859_6</td>
<td>Arabic</td>
</tr>
<tr>
<td>ISO8859_8</td>
<td>ISO8859_8</td>
<td>Hebrew</td>
</tr>
<tr>
<td>MacArabic</td>
<td>MacArabic</td>
<td>Arabic</td>
</tr>
<tr>
<td>MacHebrew</td>
<td>MacHebrew</td>
<td>Hebrew</td>
</tr>
</tbody>
</table>

-Djava.compiler

Disables the Java compiler by setting to NONE.
-Djava.compiler=[NONE | j9jit<vm_version>]
   Enable JIT compilation by setting to j9jit<vm_version> (Equivalent to -Xjit).

-Djavax.xml.namespace.QName.useCompatibleHashCodeAlgorithm
Use this property to turn off an enhanced hashing algorithm for
javax.xml.namespace.QName.hashCode().

-Djavax.xml.namespace.QName.useCompatibleHashCodeAlgorithm=1.0
   An enhanced hashing algorithm is used for
   javax.xml namespace.QName. hashCode(). This algorithm can change the
   iteration order of items returned from hash maps. For compatibility, you can
   restore the earlier hashing algorithm by setting the system property
   -Djavax.xml.namespace.QName.useCompatibleHashCodeAlgorithm=1.0.

-Djdk.map.althashing.threshold
This system property controls the use of an enhanced hashing algorithm for
hashed maps.

-Djdk.map.althashing.threshold=value
   This alternative hashing algorithm is used for string keys when a hashed data
   structure has a capacity larger than value.

   A value of 1 ensures that this algorithm is always used, regardless of the
   hashed map capacity. A value of -1 prevents the use of this algorithm, which is
   the default value.

   The hashed map structures affected by this threshold are: java.util.HashMap,
   java.util.Hashtable, java.util.LinkedHashMap, java.util.WeakHashMap, and
   java.util.concurrent.ConcurrentHashMap.

   The capacity of a hashed map is related to the number of entries in the map,
   multiplied by the load factor. Because the capacity of a hashed map is rounded
   up to the next power of two, setting the threshold to intermediate values has
   no effect on behavior. For example, threshold values of 600, 700, and 1000 have
   the same effect. However, values of 1023 and 1024 cause a difference in
   behavior. For a more detailed description of the capacity and load factor, see
   https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html

   When entries are removed from a hashed map the capacity does not shrink.
   Therefore, if the map ever exceeds the threshold to use alternative hashing for
   Strings, the map always uses alternative hashing for Strings. This behavior
does not change, even if entries are later removed or the map is emptied using
   clear().

-Djdk.reflect.allowGetCallerClass
Use this option to re-enable the sun.reflect.Reflection.getCallerClass(int depth)
   method.

-Djdk.reflect.allowGetCallerClass
   To enhance security, the sun.reflect.Reflection.getCallerClass(int depth) method
   is not supported. Use the sun.reflect.Reflection.getCallerClass() method instead.
   This method always uses a depth of 2.

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

   Note: You can use this option to re-enable support for the
   sun.reflect.Reflection.getCallerClass(int depth) method, but this option will be
removed in a future release. You must set the option on the command line when you start the application; you cannot set it from within the application at runtime.

You can use this option in several ways. The following methods enable the option:

- `-Djdk.reflect.allowGetCallerClass`
- `-Djdk.reflect.allowGetCallerClass=true`
- `-Djdk.reflect.allowGetCallerClass=true` (true is not case sensitive, so TRUE or tRuE are equally valid)

The option is disabled by default, but you can also specifically disable it by using one of the following methods:

- `-Djdk.reflect.allowGetCallerClass=false`
- `-Djdk.reflect.allowGetCallerClass=any_other_value`

**-Djdk.xml.entityExpansionLimit**
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the number of entity expansions in an XML document.

**-Djdk.xml.entityExpansionLimit=value**

where value is a positive integer. The default value is 64,000.

A value of 0 or a negative number sets no limit.

You can also set this limit by adding the following line to your jaxp.properties file:

```
jdk.xml.entityExpansionLimit=value
```

**Related reference:**

"-Djdk.xml.maxGeneralEntitySizeLimit" on page 542
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the maximum size of a general entity.

"-Djdk.xml.maxOccur" on page 543
This option provides limits for Java API for XML processing (JAXP). This option defines the maximum number of content model nodes that can be created in a grammar.

"-Djdk.xml.maxParameterEntitySizeLimit" on page 544
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the maximum size of a parameter entity.

"-Djdk.xml.maxXMLNameLimit" on page 545
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the length of XML names in XML documents.

"-Djdk.xml.totalEntitySizeLimit" on page 546
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the total size of all entities, including general and parameter entities.

"-Djdk.xml.resolveExternalEntities" on page 545
This option provides limits for Java API for XML processing (JAXP). Use this option to control whether external entities are resolved in an XML document.

**-Djdk.xml.maxGeneralEntitySizeLimit**
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the maximum size of a general entity.
To protect an application from malformed XML, set this value to the minimum size possible.

**-Djdk.xml.maxGeneralEntitySizeLimit=value**

Where *value* is the maximum size that is allowed for a general entity. The default value is 0.

A value of 0 or a negative number sets no limits.

You can also set this limit by adding the following line to your *jaxp.properties* file:

```
jdk.xml.maxGeneralEntitySizeLimit=value
```

**Related reference:**

“-Djdk.xml.entityExpansionLimit” on page 542
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the number of entity expansions in an XML document.

“-Djdk.xml.maxOccur”
This option provides limits for Java API for XML processing (JAXP). This option defines the maximum number of content model nodes that can be created in a grammar.

“-Djdk.xml.maxParameterEntitySizeLimit” on page 544
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the maximum size of a parameter entity.

“-Djdk.xml.maxXMLNameLimit” on page 545
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the length of XML names in XML documents.

“-Djdk.xml.totalEntitySizeLimit” on page 546
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the total size of all entities, including general and parameter entities.

“-Djdk.xml.resolveExternalEntities” on page 545
This option provides limits for Java API for XML processing (JAXP). Use this option to control whether external entities are resolved in an XML document.

**-Djdk.xml.maxOccur**
This option provides limits for Java API for XML processing (JAXP). This option defines the maximum number of content model nodes that can be created in a grammar.

When building a grammar for a W3C XML schema, use this option to limit the number of content model nodes that can be created when the schema defines attributes that can occur multiple times.

**-Djdk.xml.maxOccur=value**

Where *value* is a positive integer. The default value is 5,000.

A value of 0 or a negative number sets no limits.

You can also set this limit by adding the following line to your *jaxp.properties* file:

```
jdk.xml.maxOccur=value
```

**Related reference:**

“-Djdk.xml.entityExpansionLimit” on page 542
This option provides limits for Java API for XML processing (JAXP). Use this
option to limit the number of entity expansions in an XML document.

`-Djdk.xml.maxGeneralEntitySizeLimit` on page 542
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the maximum size of a general entity.

`-Djdk.xml.maxParameterEntitySizeLimit`
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the maximum size of a parameter entity.

`-Djdk.xml.maxXMLNameLimit` on page 545
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the length of XML names in XML documents.

`-Djdk.xml.totalEntitySizeLimit` on page 546
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the total size of all entities, including general and parameter entities.

`-Djdk.xml.resolveExternalEntities” on page 545
This option provides limits for Java API for XML processing (JAXP). Use this option to control whether external entities are resolved in an XML document.

**-Djdk.xml.maxParameterEntitySizeLimit**
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the maximum size of a parameter entity.

To protect an application from malformed XML, set this value to the minimum size possible.

`-Djdk.xml.maxParameterEntitySizeLimit=value`

Where value is the maximum size that is allowed for a parameter entity. The default value is 0.

A value of 0 or a negative number sets no limits.

You can also set this limit by adding the following line to your jaxp.properties file:

`jdk.xml.maxParameterEntitySizeLimit=value`

Related reference:

`-Djdk.xml.entityExpansionLimit” on page 542
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the number of entity expansions in an XML document.

`-Djdk.xml.maxGeneralEntitySizeLimit” on page 542
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the maximum size of a general entity.

`-Djdk.xml.maxOccur” on page 543
This option provides limits for Java API for XML processing (JAXP). This option defines the maximum number of content model nodes that can be created in a grammar.

`-Djdk.xml.maxXMLNameLimit” on page 545
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the length of XML names in XML documents.

`-Djdk.xml.totalEntitySizeLimit” on page 546
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the total size of all entities, including general and parameter entities.
This option provides limits for Java API for XML processing (JAXP). Use this option to control whether external entities are resolved in an XML document.

-Djdk.xml.maxXMLNameLimit
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the length of XML names in XML documents.

-Djdk.xml.maxXMLNameLimit=value
Where value is a positive integer.
A value of 0 or a negative number sets no limits. The default value is 0.

You can also set this limit by adding the following line to your jaxp.properties file:

jdk.xml.maxXMLNameLimit=value

Related reference:
-Djdk.xml.entityExpansionLimit” on page 542
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the number of entity expansions in an XML document.

-Djdk.xml.maxGeneralEntitySizeLimit” on page 542
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the maximum size of a general entity.

-Djdk.xml.maxOccur” on page 543
This option provides limits for Java API for XML processing (JAXP). This option defines the maximum number of content model nodes that can be created in a grammar.

-Djdk.xml.maxParameterEntitySizeLimit” on page 544
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the maximum size of a parameter entity.

-Djdk.xml.resolveExternalEntities
This option provides limits for Java API for XML processing (JAXP). Use this option to control whether external entities are resolved in an XML document.

-Djdk.xml.resolveExternalEntities=value
Where value is boolean. The default value is true.
A value of false turns off the resolution of XML external entities.

You can also set this limit by adding the following line to your jaxp.properties file:

jdk.xml.resolveExternalEntities=value

Related reference:
-Djdk.xml.entityExpansionLimit” on page 542
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the number of entity expansions in an XML document.
This option provides limits for Java API for XML processing (JAXP). Use this option to limit the maximum size of a general entity.

This option provides limits for Java API for XML processing (JAXP). This option defines the maximum number of content model nodes that can be created in a grammar.

This option provides limits for Java API for XML processing (JAXP). Use this option to limit the length of XML names in XML documents.

This option provides limits for Java API for XML processing (JAXP). Use this option to limit the total size of all entities, including general and parameter entities.

This option provides limits for Java API for XML processing (JAXP). Use this option to limit the maximum size of a parameter entity.

**-Djdk.xml.totalEntitySizeLimit**

This option provides limits for Java API for XML processing (JAXP). Use this option to limit the total size of all entities, including general and parameter entities.

**-Djdk.xml.totalEntitySizeLimit=value**

Where value is the collective size of all entities. The default value is \(5 \times 10^7\) \((50\ 000\ 000)\).

A value of 0 or a negative number sets no limits.

You can also set this limit by adding the following line to your jaxp.properties file:

```
jdk.xml.totalEntitySizeLimit=value
```

**Related reference:**

This option provides limits for Java API for XML processing (JAXP). Use this option to limit the number of entity expansions in an XML document.

This option provides limits for Java API for XML processing (JAXP). Use this option to limit the maximum size of a general entity.

This option provides limits for Java API for XML processing (JAXP). This option defines the maximum number of content model nodes that can be created in a grammar.

This option provides limits for Java API for XML processing (JAXP). Use this option to limit the maximum size of a parameter entity.

This option provides limits for Java API for XML processing (JAXP). Use this option to limit the length of XML names in XML documents.

This option provides limits for Java API for XML processing (JAXP). Use this option to control whether external entities are resolved in an XML document.
-Dsun.awt.keepWorkingSetOnMinimize
The -Dsun.awt.keepWorkingSetOnMinimize=true system property stops the JVM trimming an application when it is minimized.

- Dsun.awt.keepWorkingSetOnMinimize=true
When a Java application using the Abstract Windowing Toolkit (AWT) is minimized, the default behavior is to “trim” the “working set”. The working set is the application memory stored in RAM. Trimming means that the working set is marked as being available for swapping out if the memory is required by another application. The advantage of trimming is that memory is available for other applications. The disadvantage is that a “trimmed” application might experience a delay as the working set memory is brought back into RAM.

The default behavior is to trim an application when it is minimized.

-Dsun.net.client.defaultConnectTimeout
Specifies the default value for the connect timeout for the protocol handlers used by the java.net.URLConnection class.

-Dsun.net.client.defaultConnectTimeout=<value in milliseconds>
The default value set by the protocol handlers is -1, which means that no timeout is set.

When a connection is made by an applet to a server and the server does not respond properly, the applet might seem to hang. The delay might also cause the browser to hang. The apparent hang occurs because there is no network connection timeout. To avoid this problem, the Java Plug-in has added a default value to the network timeout of 2 minutes for all HTTP connections. You can override the default by setting this property.

-Dsun.net.client.defaultReadTimeout
Specifies the default value for the read timeout for the protocol handlers used by the java.net.URLConnection class when reading from an input stream when a connection is established to a resource.

-Dsun.net.client.defaultReadTimeout=<value in milliseconds>
The default value set by the protocol handlers is -1, which means that no timeout is set.

-Dsun.reflect.inflationThreshold
Controls inflation from the JNI implementation of reflection to the Java implementation of reflection.

When your application uses Java reflection, the JVM has two methods of accessing the information on the class being reflected. It can use a JNI accessor, or a Java bytecode accessor. If your application uses reflection extensively, you might want to force the JVM to use the JNI accessor because the Java bytecode accessor can use a significant amount of native memory.

-Dsun.reflect.inflationThreshold=<value>
Where a <value> sets the number of times to use the JNI accessor before the JVM changes to use the Java bytecode accessor, a process that is known as inflation. A value of 0 causes reflection never to inflate from the JNI accessor to the Java bytecode accessor.

Note: The Oracle implementation of this system property is different. Setting the value to 0 causes reflection to inflate from the JNI implementation of
reflection to the Java implementation of reflection after the first usage. If you want to force the use of the Java implementation of reflection, use 
-Dsun.reflect.noInflation=true.

-Dsun.rmi.transport.tcp.connectionPool
Enables thread pooling for the RMI ConnectionHandlers in the TCP transport layer implementation.

-Dsun.rmi.transport.tcp.connectionPool=val
val is either true or a value that is not null.

-Dswing.useSystemFontSettings
This option addresses compatibility problems for Swing programs.

-Dswing.useSystemFontSettings=[false]
By default, Swing programs running with the Windows Look and Feel render with the system font set by the user instead of a Java-defined font. As a result, fonts differ from the fonts in earlier releases. This option addresses compatibility problems like these for programs that depend on the old behavior. By setting this option, v1.4.1 fonts and those of earlier releases are the same for Swing programs running with the Windows Look and Feel.

-Dorg.omg.CORBA.ORBid
This system property uniquely identifies an ORB in its address space.

-Dorg.omg.CORBA.ORBid=<string>
Where string is a unique identifier. The default value is a randomly generated number that is unique in the JVM to the ORB.

-Dorg.omg.CORBA.ORBListenEndpoints
This system property identifies the set of endpoints on which the ORB listens for requests. Endpoints consist of a hostname or IP address, and optionally a port.

-Dorg.omg.CORBA.ORBListenEndpoints=<value>
Where value is a string of the form hostname:portnumber, where the :portnumber component is optional. IPv6 addresses must be surrounded by brackets (for example, [::1]:1020). Specify multiple endpoints in a comma-separated list.

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

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

-Dorg.omg.CORBA.ORBServerId
This system property identifies the ORB server. Assign the same value for this property to all ORBs contained in the same server. It is included in all Interoperable Object References (IORs) exported by the server.

-Dorg.omg.CORBA.ORBServerId=<value>
Where value is in the range 0 - 2147483647.
**JVM command-line options**

Use these options to configure your JVM. The options prefixed with `-X` are nonstandard.

Options that relate to the JIT are listed under "JIT and AOT command-line options" on page 573. Options that relate to the Garbage Collector are listed under "Garbage Collector command-line options" on page 577.

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.

- `-X`
  Displays help on nonstandard options.

- `-X` Displays help on nonstandard options.

- `-Xaggressive`
  Enables performance optimizations.

  `-Xaggressive`
  Enables performance optimizations and new platform exploitation that are expected to be the default in future releases.

- `-Xargencoding`
  Include Unicode escape sequences in the argument list.

  `-Xargencoding`
  You can use the Unicode escape sequences in the argument list that you pass to this option. To specify a Unicode character, use escape sequences in the form `\u####`, where `#` is a hexadecimal digit (0 - 9, A to F).

  `-Xargencoding:utf8`
  Use utf8 encoding.

  `-Xargencoding:latin`
  Use ISO8859_1 encoding.

To specify a class that is called HelloWorld and use Unicode encoding for both capital letters, specify this command:

`java -Xargencoding '\u0048ello\u0057orld'`

- `-Xbootclasspath`
  Sets the search path for bootstrap classes and resources.

  `-Xbootclasspath:<directories and compressed or Java archive files separated by : (; on Windows)>`
  The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

- `-Xbootclasspath/a: `
  Appends to the end of the search path for bootstrap classes.

  `-Xbootclasspath/a:<directories and compressed or Java archive files separated by : (; on Windows)>`
  Appends the specified directories, compressed files, or .jar files to the end of the bootstrap class path. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.
-Xbootclasspath/p:

Adds a prefix to the search path for bootstrap classes.

-Xbootclasspath/p:<directories and compressed or Java archive files separated by : (; on Windows)>

Adds a prefix of the specified directories, compressed files, or Java archive files to the front of the bootstrap class path. Do not deploy applications that use the -Xbootclasspath: or the -Xbootclasspath/p: option to override a class in the standard API. The reason is that such a deployment contravenes the Java 2 Runtime Environment binary code license. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xcheck

You can use the -Xcheck option to run checks during JVM startup, such as memory checks or checks on JNI functions.

-Xcheck:<option>

The options available are detailed in separate topics.

-Xcheck:classpath:

Displays a warning message if an error is discovered in the class path.

-Xcheck:classpath

Checks the classpath and reports if an error is discovered; for example, a missing directory or JAR file.

-Xcheck:dump:

Runs checks on operating system settings during JVM startup.

-Xcheck:dump

Messages are issued if the operating system has dump options or limits set that might truncate system dumps.

The following messages are possible:

JVMJ9VM133W The system core size hard ulimit is set to <value>, system dumps may be truncated

This message indicates that the AIX operating system user limit is set to restrict the size of system dumps to the value indicated. If a system dump is produced by the JVM it might be truncated, and therefore of greatly reduced value in investigating the cause of crashes and other issues. For more information on how to set user limits on AIX, see “Enabling full AIX core files” on page 256.

JVMJ9VM134W The system fullcore option is set to FALSE, system dumps may be truncated

This message indicates that the AIX operating system Enable full CORE dump option is set to FALSE. This setting might result in truncated system dumps. For more information about how to set this option correctly on AIX, see “Enabling full AIX core files” on page 256.

-Xcheck:gc:

Runs additional checks on garbage collection.
By default, no checks are made. See the output of \texttt{-Xcheck:gc\:help} for more information.

\textbf{-Xcheck:gc}[[:<scan options>][[:<verify options>][[:<misc options>]]]

- \textbf{-Xcheck:jni:}

Runs additional checks for JNI functions.

\textbf{-Xcheck:jni[[:<option>=<value>]]}

This option is equivalent to \texttt{-Xrunjnichk}. By default, no checks are made.

\textbf{-Xcheck:memory:}

Identifies memory leaks inside the JVM.

\textbf{-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, \texttt{all} is used by default. The available options are as follows:

\texttt{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.

\texttt{callsite=<number of allocations>}

Displays callsite information every \texttt{<number of allocations>}. De-allocations are not counted. Callsite information is presented in a table with separate information for each callsite. Statistics include:

- The number and size of allocation and free requests since the last report.
- The number of the allocation request responsible for the largest allocation from each site.

Callsites are presented as \texttt{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.

\texttt{failat=<number of allocations>}

Causes memory allocation to fail (return NULL) after \texttt{<number of allocations>}. Setting \texttt{<number of allocations>} to 13 causes the 14th allocation to return NULL. De-allocations are not counted. Use this option to ensure that JVM code reliably handles allocation failures. This option is useful for checking allocation site behavior rather than setting a specific allocation limit.

\texttt{ignoreUnknownBlocks}

Ignores attempts to free memory that was not allocated using the \texttt{-Xcheck:memory} tool. Instead, the \texttt{-Xcheck:memory} statistics that are printed out at the end of a run indicates the number of “unknown” blocks that were freed.

\texttt{mprotect=<top|bottom>}

Locks pages of memory on supported platforms, causing the program to stop if padding before or after the allocated block is accessed for reads or writes. An extra page is locked on each side of the block returned to the user.
If you do not request an exact multiple of one page of memory, a region on one side of your memory is not locked. The top and bottom options control which side of the memory area is locked. top aligns your memory blocks to the top of the page (lower address), so buffer underruns result in an application failure. bottom aligns your memory blocks to the bottom of the page (higher address) so buffer overruns result in an application failure.

Standard padding scans detect buffer underruns when using top and buffer overruns when using bottom.

**nofree**

Keeps a list of blocks that are already used instead of freeing memory. This list, and the list of currently allocated blocks, is checked for memory corruption on every allocation and deallocation. Use this option to detect a dangling pointer (a pointer that is "dereferenced" after its target memory is freed). This option cannot be reliably used with long-running applications (such as WebSphere Application Server), because “freed” memory is never reused or released by the JVM.

**noscan**

Checks for blocks that are not freed. This option has little effect on performance, but memory corruption is not detected. This option is compatible only with subAllocator, callsite, and callsitesmall.

**quick**

Enables block padding only and is used to detect basic heap corruption. Every allocated block is padded with sentinel bytes, which are verified on every allocate and free. Block padding is faster than the default of checking every block, but is not as effective.

**skipto=**<number of allocations>

Causes the program to check only on allocations that occur after <number of allocations>. De-allocations are not counted. Use this option to speed up JVM startup when early allocations are not causing the memory problem. The JVM performs approximately 250+ allocations during startup.

**subAllocator[=size in MB]**

Allocates a dedicated and contiguous region of memory for all JVM allocations. This option helps to determine if user JNI code or the JVM is responsible for memory corruption. Corruption in the JVM subAllocator heap suggests that the JVM is causing the problem; corruption in the user-allocated memory suggests that user code is corrupting memory. Typically, user and JVM allocated memory are interleaved.

**zero**

Newly allocated blocks are set to 0 instead of being filled with the \(0xE7E7xxxxxxxxE7E7\) pattern. Setting these blocks to 0 helps you to determine whether a callsite is expecting zeroed memory, in which case the allocation request is followed by memset(pointer, 0, size).

**Note:** The `-Xcheck:memory` option cannot be used in the `-Xoptionsfile`.

**-Xcheck:vm:**

Performs additional checks on the JVM.

**-Xcheck:vm[:<option>]**

By default, no checking is performed. For more information, run `-Xcheck:vm:help`. 
-Xclassgc
Enables dynamic unloading of classes by the JVM. Garbage collection of class objects occurs only on class loader changes.

-XXclassgc
Dynamic unloading is the default behavior. To disable dynamic class unloading, use the -Xnoclassgc option.

-Xcompressedrefs
Enables the use of compressed references.

-Xcompressedrefs
(64-bit only) To disable compressed references, use the -Xnoclassgc option. For more information, see “Compressed references” on page 73.

Compressed references are enabled by default on all platforms when the value of the -Xmx option is less than or equal to 25 GB.

You cannot include this option in an options file. You must specify this option on the command line, or by using the IBM_JAVA_OPTIONS environment variable.

-Xdiagnosticscollector
This option is now redundant, and only generates a warning message.

-Xdiagnosticscollector[:settings=<filename>]
The Diagnostics Collector was a utility provided in previous releases. This utility has now been removed; instead, use the IBM Support Assistant Data Collector. For more information, see “IBM Support Assistant Data Collector” on page 436.

-Xdisablejavadump
Turns off Javadump generation on errors and signals.

-Xdisablejavadump
By default, Javadump generation is enabled.

-Xdump
Use the -Xdump option to add and remove dump agents for various JVM events, update default dump settings (such as the dump name), and limit the number of dumps that are produced.

-Xdump
See “Using dump agents” on page 321 for more information.

-Xenableexplicitgc
This options tells the VM to trigger a garbage collection when a call is made to System.gc().

-Xenableexplicitgc
Signals to the VM that calls to System.gc() trigger a garbage collection. This option is enabled by default.

-Xfastresolve
Tune performance by improving the resolution time for classes.

-Xfastresolve
This option is used to tune performance by improving the resolution time for classes when the field count exceeds the threshold specified by <n>. If profiling
tools show significant costs in field resolution, change the threshold until the costs are reduced. If you enable this option, additional memory is used when the threshold is exceeded.

-Xfuture
Turns on strict class-file format checks.

-Xfuture
Use this flag when you are developing new code because stricter checks will become the default in future releases. By default, strict format checks are disabled.

-Xiss
Sets the initial stack size for Java threads.

-Xiss<size>
By default, the stack size is set to 2 KB. Use the -verbose:sizes option to output the value that the VM is using.

-Xjarversion
Produces output information about the version of each .jar file.

-Xjarversion
Produces output information about the version of each .jar file in the class path, the boot class path, and the extensions directory. Version information is taken from the Implementation-Version and Build-Level properties in the manifest of the .jar file.

Note: The -Xjarversion option cannot be used in the -Xoptionsfile.

-Xjni
Sets JNI options.

-Xjni:<suboptions>
You can use the following suboption with the -Xjni option:

-Xjni:arrayCacheMax=[<size in bytes>|unlimited]
Sets the maximum size of the array cache. The default size is 128 KB.

-Xlinenumbers
Displays line numbers in stack traces for debugging.

-Xlinenumbers
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.

-XlockReservation
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.

-Xlockword
Test whether performance optimizations are negatively impacting an application.

-Xlockword:<options>

-Xlockword:[mode=all|mode=default]
See “Testing JVM optimizations” on page 289.
-Xlockword:unlockword=<class_name>
This option removes the lockword from object instances of the class
<class_name>, reducing the space required for these objects. However, this
action might have an adverse effect on synchronization for those objects.
You should not use this option unless you are directed to by IBM service.

-Xlog
Enables message logging.
-Xlog[:help]|[:<option>]
Optional parameters are:
* help - details the options available
* error - turns on logging for all JVM error messages (default).
* vital - turns on logging for selected information messages JVMDUMP006I,
  JVMDUMP032I, and JVMDUMP033I, which provide valuable additional
  information about dumps produced by the JVM (default).
* info - turns on logging for all JVM information messages
* warn - turns on logging for all JVM warning messages
* config - turns on logging for all JVM configuration messages
* all - turns on logging for all JVM messages
* none - turns off logging for all JVM messages

Note: Changes made to message logging using the -Xlog option do not affect
messages written to the standard error stream (stderr).
The options all, none and help must be used on their own and cannot be
combined. However, the other options can be grouped. For example, to include
error, vital and warning messages use -Xlog: error, vital, warn. For message
details see "JVM messages" on page 595.

-Xlp
Requests the JVM to allocate the Java object heap and JIT code cache memory with
large pages.
-Xlp[<size>]
AIX: Requests the JVM to allocate the Java object heap (the heap from which
Java objects are allocated) with large (16 MB) pages, if a size is not specified. If
large pages are not available, the Java object heap is allocated with the next
smaller page size that is supported by the system. AIX requires special
configuration to enable large pages. For more information about configuring
AIX support for large pages, see [Large pages] in the AIX product
documentation. The SDK supports the use of large pages only to back the Java
object heap shared memory segments. The JVM uses shmget() with the
SHM_LGPG and SHM_PIN flags to allocate large pages. The -Xlp option
replaces the environment variable IBM_JAVA_LARGE_PAGE_SIZE, which is now
ignored if set.

For more information, see "Configuring large page memory allocation" on
page 231.

AIX, Linux, Windows: If a <size> is specified, the JVM attempts to allocate the
JIT code cache memory using pages of that size. Allocating large pages using
-Xlp[<size>] is only supported on the 64-bit SDK, not the 32-bit SDK.
**All platforms:** To obtain the large page sizes available and the current setting, use the `-verbose:sizes` option. Note the current settings are the requested sizes and not the sizes obtained. For object heap size information, check the `-verbose:gc` output.

The JVM ends if there are insufficient operating system resources to satisfy the request. However, an error message is not issued. This limitation and a workaround for verifying the page size that is used can be found in Known limitations.

For more information, see “Configuring large page memory allocation” on page 231.

**-Xlp:codecache:**

Requests the JVM to allocate the JIT code cache by using large page sizes.

**-Xlp:codecache:pagesize=<size>** *(AIX, Linux, and Windows)*

If the requested large page size is not available, the JVM starts, but the JIT code cache is allocated using a platform-defined size. A warning is displayed when the requested page size is not available.

To obtain the large page sizes available and the current setting, use the `-verbose:sizes` option. Note the current settings are the requested sizes and not the sizes obtained.

For more information, see “Configuring large page memory allocation” on page 231.

**AIX:** The code cache page size is controlled by the DATAPSIZE setting of the LDR_CNTRL environment variable. The page size cannot be controlled by the `-Xlp:codecache:pagesize=<size>` option. Specifying any other page size results in a warning that the page size is not available. The `-verbose:sizes` output reflects the current operating system setting. For more information about the LDR_CNTRL environment variable, see “Working with the LDR_CNTRL environment variable” on page 212.

**-Xlp:objectheap:**

Requests the JVM to allocate the Java object heap by using large page sizes.

**-Xlp:objectheap:pagesize=<size>,[strict],[warn]**

Where:
- `<size>` is the large page size that you require.
- `strict` causes an error message to be generated if large pages are requested but cannot be obtained. This suboption causes the JVM to end.
- `warn` causes a warning message to be generated if large pages are requested but cannot be obtained. This suboption allows the JVM to continue.

**Note:** If both suboptions are specified, strict overrides warn.

If the operating system does not have sufficient resources to satisfy the request, the page size you requested might not be available when the JVM starts up. By default, the JVM starts and the Java object heap is allocated by using a different platform-defined page size. Alternatively, you can use the strict or warn suboptions to customize behavior.

If you are running an earlier version that does not include the strict or warn suboptions, an error message is not generated when there are insufficient
resources available. This limitation and a workaround for verifying which page size is used can be found in “Known issues and limitations” on page 606.

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

For more information, see “Configuring large page memory allocation” on page 231.

-Xmso
Sets the initial stack size for operating system threads.

-Xmso<size>
The default value can be determined by running the command:
java -verbose:sizes

The maximum value for the stack size varies according to platform and specific machine configuration. If you exceed the maximum value, a java/lang/StackOverflowError message is reported.

-Xnoagent
Disables support for the old JDB debugger.

-Xnoagent
Disables support for the old JDB debugger.

-Xnoclassgc
Disables class garbage collection.

-Xnoclassgc
This option switches off garbage collection of storage associated with Java technology 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 support team. The reason is the option can cause unlimited native memory growth, leading to out-of-memory errors.

-Xnocompressedrefs
Disables the use of compressed references.

-Xnocompressedrefs
(64-bit only)
This option disables the use of compressed references.

You cannot include this option in an options file. You must specify this option on the command line, or by using the IBM_JAVA_OPTIONS environment variable.

To enable compressed references, use the -Xcompressedreferences option. For more information, see “Compressed references” on page 73.

-Xnolinenumbers
Disables the line numbers for debugging.

-Xnolinenumbers
See also -Xlinenumbers. By default, line number are on.

If you start the JVM with -Xnolinenumbers when creating a new shared classes cache, the Class Debug Area is not created. The option -Xnolinenumbers advises the JVM not to load any class debug information, so there is no need
for this region. If \texttt{-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 \texttt{-Xnolinenumbers}.

\textbf{-Xnosigcatch}

Disables JVM signal handling code.

\textbf{-Xnosigcatch}

See also \texttt{-Xsigcatch}. By default, signal handling is enabled.

\textbf{-Xnosigchain}

Disables signal handler chaining.

\textbf{-Xnosigchain}

See also \texttt{-Xsigchain}. By default, the signal handler chaining is enabled.

\textbf{-Xoptionsfile}

Specifies a file that contains VM options and definitions. The contents of the options file are recorded in the EWINF0 section of a Java dump.

\texttt{-Xoptionsfile=\textless file\textgreater}

where \textless file\textgreater contains options that are processed as if they had been entered directly as command-line options. At startup, the VM automatically adds \texttt{-Xoptionsfile=\$\textless vm\_dir\textgreater/\texttt{options.default}} at the beginning of the command line, where \texttt{\$\textless vm\_dir\textgreater} is the path specified in \textit{"Conventions and terminology" on page 62}. The file \texttt{options.default} is empty but can be updated with any options that you want to specify at run time.

Here is an example of an options file:

\begin{verbatim}
#My options file
-X<option1>
-X<option2>=\<value1>,<value2>
-D<sysprop1>=<value1>
\end{verbatim}

The options file does not support these options:

\begin{itemize}
  \item \texttt{-assert}
  \item \texttt{-fullversion}
  \item \texttt{-help}
  \item \texttt{-showversion}
  \item \texttt{-version}
  \item \texttt{-Xcompressedrefs}
  \item \texttt{-Xcheck:memory}
  \item \texttt{-Xjarversion}
  \item \texttt{-Xoptionsfile}
\end{itemize}

Although you cannot use \texttt{-Xoptionsfile} recursively within an options file, you can use \texttt{-Xoptionsfile} multiple times on the same command line to load more than one options files.

Some options use quoted strings as parameters. Do not split quoted strings over multiple lines using the forward slash line continuation character (\). The Yen symbol (¥) is not supported as a line continuation character. For example, the following example is not valid in an options file:

\begin{verbatim}
-Xevents=vmstop,exec="cmd /c \ echo \%pid has finished."
\end{verbatim}
The following example is valid in an options file:

```bash
-Xevents=vmstop, \n  exec="cmd /c echo %pid has finished."
```

**Related information:**

- "Specifying command-line options” on page 525
- "TITLE, GPINFO, and ENVINFO sections” on page 343

Although the command line is the traditional way to specify command-line options, you can also pass options to the Java virtual machine (VM) by using a manifest file, options files, and environment variables.

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.

**-Xrdbginfo**

Loads the remote debug information server with the specified host and port.

```
-Xrdbginfo:<host>[:<port>]
```

By default, the remote debug information server is disabled.

**-Xrs**

Disables signal handling in the JVM.

Setting **-Xrs** prevents the Java runtime environment from handling any internally or externally generated signals such as SIGSEGV and SIGABRT. Any signals that are raised are handled by the default operating system handlers. Disabling signal handling in the JVM reduces performance by approximately 2-4%, depending on the application.

**-Xrs:sync**

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.

**Note:** Setting this option prevents dumps being generated by the JVM for signals such as SIGSEGV and SIGABRT, because the JVM is no longer intercepting these signals.

**-Xscdmx**

Use the **-Xscdmx** option to control the size of the class debug area when you create a shared class cache.

```
-Xscdmx<size>
```

The **-Xscdmx** option works in a similar way to the **-Xscmx** option, which is used to control the overall size of the shared class cache. The size of **-Xscdmx** must be smaller than the size of **-Xscmx**. By default, the size of the class debug area is a percentage of the free class data bytes in a newly created or empty cache.

_size_ is expressed as an absolute value; number of bytes, kilobytes (512k), megabytes (2m), or gigabytes (1g).

A class debug area is still created if you use the **-Xnolinenumbers** option with the **-Xscdmx** option on the command line.
**-Xscmx**
Specifies the size of a new cache.

**-Xscmx<size>**
This option applies only if a cache is being created and no cache of the same name exists.

The default cache size is platform-dependent. You can find out the size value being used by adding **-verbose:sizes** as a command-line argument. The minimum cache size is 4 KB. The 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. In limited memory environments, such as when running a 32-bit VM, increasing the maximum size of the Java heap reduces the size of the shared classes cache that you can create.

For more information about cache sizes, see "Cache size limits" on page 244.

**-Xshareclasses**
Enables class sharing. This option can take a number of suboptions, some of which are cache utilities.

**-Xshareclasses:<suboptions>**
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.

**Note:** When running cache utilities, the message **Could not create the Java virtual machine** is expected. Cache utilities do not create the virtual machine. Some cache utilities can work with caches from previous Java versions or caches that are created by JVMs with different bit-widths. These caches are referred to as "incompatible" caches.

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

**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`.

**Note:** From service refresh 2: If you specify this option, persistent caches are created with read/write permission for the user, read-only permission for others, and read-only or read/write permission for groups depending on whether you also specify the **-Xshareclasses:groupAccess** option. Otherwise, persistent caches are created with the same permissions as
non-persistent caches: read/write permission for the user, and read/write permission for the group depending on whether you also specify the
-Xshareclasses:groupAccess option.

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 ends with an appropriate error message.
The permissions specified by this suboption are used only when creating a new cache directory. If the cache directory already exists, this suboption is ignored and the cache directory permissions are not changed.
If you set this suboption to 0000, the default directory permissions are used. If you set this suboption to 1000, the machine default directory permissions are used, but the sticky bit is enabled. If the cache directory is the platform default directory, /tmp/javasharedresources, this suboption is ignored and the cache directory permissions are set to 777. If you do not set this suboption, the cache directory permissions are set to 777, for compatibility with earlier Java versions.

cacheRetransformed
Enables caching of classes that are transformed by using the JVMTI RetransformClasses function. See "JVMTI redefinition and retransformation of classes" on page 474 for more information.
The option enableBCI is enabled by default. However, if you use the cacheRetransformed option, this option forces cache creation into -Xshareclasses:disableBCI mode.

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

destroyAll (Utility option)
Tries to destroy all caches available using the specified cacheDir and nonpersistent suboptions. A non-persistent cache can be destroyed only if all JVMs that are using it have shut down and the user has sufficient permissions. Persistent caches that are still in use continue to exist even when you use this option, but they are unlinked from the file system so they are not visible to new JVM invocations. If you update the JVM then restart an application for which a persistent shared cache already exists, the JVM unlinks the existing cache and creates a new cache. Because the unlinked caches are not visible to new JVMs, you cannot find them by using the -Xshareclasses:listAllCaches option, and you cannot use the -Xshareclasses:printStats option on them. You can therefore have multiple unlinked caches that consume file system space until they are no longer in use.

disableBCI
Turns off BCI support. This option can be used to override -XX:ShareClassesEnableBCI. For more information, see "JVM -XX:command-line options" on page 569.

enableBCI
This option is enabled by default.
Allows a JVMTI ClassFileLoadHook event to be triggered every time, for classes that are loaded from the cache. This mode also prevents caching of
classes that are modified by JVMTI agents. For more information about this
option, see “Using the JVMTI ClassFileLoadHook with cached classes” on
page 473. This option is incompatible with the cacheRetransformed option.
Using the two options together causes the JVM to end with an error
message, unless -Xshareclasses:nonfatal is specified. In this case, the
JVM continues without using shared classes.

A cache that is created without the enableBCI suboption cannot be reused
with the enableBCI suboption. Attempting to do so causes the JVM to end
with an error message, unless -Xshareclasses:nonfatal is specified. In this
case, the JVM continues without using shared classes. A cache that is
created with the enableBCI suboption can be reused without specifying this
suboption. In this case, the JVM detects that the cache was created with the
enableBCI suboption and uses the cache in this mode.

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

**findAotMethods=help|{<method_specification> [, <method_specification>]}
(Utility option)
Print the AOT methods in the shared cache that match the method
specifications. Methods that are already invalidated are indicated in the
output. Use this suboption to check which AOT methods in the shared
class cache would be invalidated by using the same method specifications
with the invalidateAotMethods suboption. To learn more about the syntax
to use when you are specifying more than one method specification, see
“Method specification syntax” on page 566.

**groupAccess**
Sets operating system permissions on a new cache to allow group access to
the cache. Group access can be set only when permitted by the operating
system umask setting. The default is user access only.

From service refresh 2: If a user creates a cache by specifying the
groupAccess suboption, other users in the same group must also specify
this suboption to be granted access to the same cache.

**help**
Lists all the command-line options.

**invalidateAotMethods=help|{<method_specification> [, <method_specification>]}
(utility option)
Modify the existing shared cache to invalidate the AOT methods matching
the method specifications. Use this suboption to invalidate AOT methods
that cause a failure in the application, without having to destroy the shared
cache. Invalidated AOT methods remain in the shared cache, but are then
excluded from being loaded. JVMs that have not processed the methods, or
new JVMs that use the cache are not affected by the invalidated methods.
The AOT methods are invalidated for the lifetime of the cache, but do not
prevent the AOT methods from being compiled again if a new shared
cache is created. To prevent AOT method compilation into a new shared
cache, use the -Xaot:exclude option. For more information, see “-Xaot”
on page 573. To identify AOT problems, see “Diagnosing a JIT or AOT
problem” on page 430. To revalidate an AOT method, see the
revalidateAotMethods suboption. Use the findAotMethod suboption to
determine which AOT methods match the method specifications. To learn
more about the syntax to use when you are specifying more than one method specification, see “Method specification syntax” on page 566.

**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=[default | all | none]**
Where:
- default: By default, the memory pages that contain the cache are always protected, unless a specific page is being updated. This protection helps prevent accidental or deliberate corruption to the cache. The cache header is not protected by default because this protection has a performance cost.
- all: This option ensures that all the cache pages are protected, including the header.
- none: Specifying this option disables the page protection.

*Note:* Specifying all has a negative impact on performance. You should specify all only for problem diagnosis and not for production.

**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 471 for more information.

If you are migrating from IBM SDK, Java Technology Edition, Version 7, or earlier releases, you must set `disableBCI` when using this option to retain the same behavior. For more information see, `disableBCI`.

**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. AOT code already in the shared data cache can be loaded.

**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 479 for more information about shared class filtering.
nojitdata
Disables caching of JIT data. JIT data already in the shared data cache can be loaded.

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. This suboption disables the shared class utility APIs. To disable class data sharing without disabling the utility APIs, use the utilities suboption. For more information about the shared class utility APIs, see “Obtaining information about shared caches” on page 480.

nonfatal
Allows the JVM to start even if class data sharing fails. Normal behavior for the JVM is to refuse to start if class data sharing fails. If you select nonfatal and the shared classes cache fails to initialize, the JVM attempts to connect to the cache in read-only mode. If this attempt fails, the JVM starts without class data sharing.

nonpersistent
Uses a nonpersistent cache. The cache is lost when the operating system shuts down. Nonpersistent and persistent caches can have the same name.

persistent (default for AIX, Windows, and Linux platforms)
Uses a persistent cache. The cache is created on disk, which persists beyond operating system restarts. Nonpersistent and persistent caches can have the same name. You must always use the persistent suboption when running utilities such as destroy on a persistent cache.

printAllStats (Utility option)
Displays detailed information about the contents of the cache that is specified in the name=<name> suboption. If the name is not specified, statistics are displayed about the default cache. Every class is listed in chronological order with a reference to the location from which it was loaded. See “printAllStats utility” on page 486 for more information.

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

Specify one or more data types, which are separated by a plus symbol (+), to additionally see more detailed information about the cache content. Data types include AOT data, class paths, and ROMMethods. See “printStats utility” on page 482 for more information.

rcdSize=nnn
This option is deprecated in IBM SDK, Java Technology Edition, Version 7 Release 1, and will be removed from future releases. This option is ignored, if used.

readonly
Opens an existing cache with read-only permissions. The Java virtual machine does not create a new cache with this suboption. Opening a cache read-only prevents the VM from making any updates to the cache. If you specify this suboption, the VM can connect to caches that were created by other users or groups without requiring write access. However, from
service refresh 2, this access is permitted only if the cache was created by
using the `-Xshareclasses:cacheDir` option to specify a directory with
appropriate permissions. If you do not use the `-Xshareclasses:cacheDir`
option, the cache is created with default permissions, which do not permit
access by other users or groups.

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.

**revalidateAotMethods=help|{"<method_specification>"[,"<method_specification>"]}**
(Utility option)
Modify the shared cache to revalidate the AOT methods that match the
method specifications. Use this suboption to revalidate AOT methods that
were invalidated by using the `invalidateAotMethods` suboption.
Revalidated AOT methods are then eligible for loading into a JVM, but do
not affect JVMs where the methods have already been processed. To learn
more about the syntax to use when you are specifying more than one
method specification, see "Method specification syntax" on page 566.

**silent**
Disables all shared class messages, including error messages.
Unrecoverable error messages, which prevent the JVM from initializing, are
displayed.

**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 480.

**verbose**
Gives detailed output on the cache I/O activity, listing information about
classes that are stored and found. Each class loader is given a unique ID
(the bootstrap loader is always 0) and the output shows the class loader
hierarchy at work, where class loaders must ask their parents for a class
before they can load it themselves. It is typical to see many failed requests;
this behavior is expected for the class loader hierarchy. The standard
option `-verbose:class` also enables class sharing verbose output if class
sharing is enabled.

**verboseAOT**
Enables verbose output when compiled AOT code is being found or stored
in the cache. AOT code is generated heuristically. You might not see any
AOT code that is generated at all for a small application. You can disable
AOT caching using the `noaot` suboption. See the IBM JVM Messages Guide
for a list of the messages produced.

**verboseHelper**
Enables verbose output for the Java Helper API. This output shows you
how the Helper API is used by your class loader.

**verboseIO**
Gives detailed output on the cache I/O activity, listing information about
classes that are stored and found. Each class loader is given a unique ID
(the bootstrap loader is always 0) and the output shows the class loader
hierarchy at work, where class loaders must ask their parents for a class before they can load it themselves. It is typical to see many failed requests; this behavior is expected for the class loader hierarchy.

**Method specification syntax**

The following examples show how to specify more than one method specification when you are using the `findAotMethods=`, `invalidateAotMethods=`, or `revalidateAotMethods=` suboptions.

Braces, `{}`, are required around the method specification if you specify more than one method specification. If the specification contains a comma, `<method_specification>` is defined as the following string:

```
[!]{[*]{packagename/classname}[*].[{[*]{methodname}[*]}]}{(*)[*]{parameters}[*]}]
```

Parameters are optional, but if specified, the following native signature formats must be used:

- B for byte
- C for char
- D for double
- F for float
- I for int
- J for long
- S for short
- Z for Boolean
- L<class name>; for objects
- [ before the signature means array

If parameters must be specified to distinguish the method, use `findAotMethods=` with the string (*) to list all the parameter variations. Copy the signature for the method that you want from the output. For example, the signature for the parameters `(byte[] bytes, int offset, int length, Charset charset)` is `([BIILjava/nio/charset/Charset;)`.

Here are some examples:

- * - matches all AOT methods.
- java/lang/Object - matches all AOT methods in the java.lang.Object class.
- java/util/* - matches all AOT classes and methods in the java.util package.
- java/util/HashMap.putVal - matches all putVal methods in the java.util.HashMap class.
- *.equals - matches all equals methods in all classes.
- {java/util/*,java/lang/Object.*(*)} - matches all classes or methods with no input parameter in the java.util package, and all methods in java.lang.Object.
- {java/util/*,!java/util/*()} - matches nothing.

-**-Xsigcatch**

Enables VM signal handling code.
-Xsigcatch
  See also -Xnosigcatch. By default, signal handling is enabled.

-XX:sigchain
  Enables signal handler chaining.

-XX:sigchain
  See also -Xnosigchain. By default, signal handler chaining is enabled.

-XXs
  Sets the maximum stack size for Java threads.

-XXs<size>
  The default is 320 KB for 31-bit or 32-bit JVMs and 1024 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.

-XXssi
  Sets the stack size increment for Java threads.

-XXssi<size>
  When the stack for a Java thread becomes full it is increased in size by this value until the maximum size (-XXs) is reached. The default is 16 KB.

-XXthr
  -XXthr:<suboptions>

    -XXthr:<AdaptSpin|noAdaptSpin>
    This tuning option is available to test whether performance optimizations are negatively impacting an application. See “Testing JVM optimizations” on page 289.

    -XXthr:minimizeUserCPU
    Minimizes user-mode CPU usage in thread synchronization where possible. The reduction in CPU usage might be a trade-off in exchange for decreased performance.

    -XXthr:<secondarySpinForObjectMonitors|noSecondarySpinForObjectMonitors>
    This tuning option is available to test whether performance optimizations are negatively impacting an application. See “Testing JVM optimizations” on page 289.

-XXtlhPrefetch
  Speculatively prefetches bytes in the thread local heap (TLH) ahead of the current allocation pointer during object allocation.

-XXtlhPrefetch
  This option helps reduce the performance cost of subsequent allocations.

-XXtrace
  Trace options.

-XXtrace[:help] [:<option>=<value>, ...]
  See “Controlling the trace” on page 398 for more information.

-XXtune:virtualized
  Optimizes JVM function for virtualized environments, such as a cloud.

-XXtune:virtualized
  Optimizes JVM function for virtualized environments, such as a cloud.
-Xverify
Use this option to enable or disable the verifier.

-Xverify[:<option>]
With no parameters, enables the verifier, which is the default. Therefore, if used on its own with no parameters, for example, -Xverify, this option does nothing. Optional parameters are as follows:
• all  - enable maximum verification
• none  - disable the verifier
• remote  - enables strict class-loading checks on remotely loaded classes

The verifier is on by default and must be enabled for all production servers. Running with the verifier off is not a supported configuration. If you encounter problems and the verifier was turned off using -Xverify:none, remove this option and try to reproduce the problem.

-Xzero
Enables reduction of the memory footprint of the Java runtime environment when concurrently running multiple Java invocations. This option is deprecated since IBM SDK, Java Technology Edition, Version 7 Release 1, and will be removed from a future release.

-Xzero[:<option>]
-Xzero might not be appropriate for all types of applications because it changes the implementation of java.util.ZipFile, which might cause extra memory usage. -Xzero includes the optional parameters:
• none  - disables all sub options
• describe  - prints the sub options in effect
• sharebootzip  - enables the sharebootzip sub option
• nosharebootzip  - disables the sharebootzip sub option
• j9zip  - enables the j9zip sub option
• noj9zip  - disables the j9zip sub option
• sharezip  - enables the sharezip sub option
• nosharezip  - disables the sharezip sub option

Because future versions might include more default options, -Xzero options are used to specify the sub options that you want to disable. By default, -Xzero enables j9zip and sharezip. A combination of j9zip and sharezip enables all .jar files to have shared caches:
• j9zip  - uses a new java.util.ZipFile implementation. This suboption is not a requirement for sharezip; however, if j9zip is not enabled, only the bootstrap .jar files have shared caches.
• sharezip  - puts the j9zip cache into shared memory. The j9zip cache is a map of zip entry names to file positions used to quickly find entries in the .zip file. You must enable -Xshareclasses to avoid a warning message. When using the sharezip suboption, note that this suboption allows every opened .zip file and .jar file to store the j9zip cache in shared memory, so you might fill the shared memory when opening multiple new .zip files and .jar files. The affected API is java.util.zip.ZipFile (superclass of java.util.jar.JarFile). The .zip and .jar files do not have to be on a class path.
• sharebootzip  - enabled by default on all platforms. Puts the zip entry caches for bootstrap .jar files into the shared cache. A zip entry cache is a map of zip entry names to file positions, used to quickly find entries in the .zip file.
The system property com.ibm.zeroversion is defined, and has a current value of 2. Although -Xzero is accepted on all platforms, support for the sub options varies by platform:

- -Xzero with the sharebootzip and nosharebootzip sub options are accepted on all platforms.
- -Xzero with all other sub options are available only on Windows x86-32 and Linux x86-32 platforms.

### JVM -XX: command-line options

Java VM command-line options that are specified with -XX: are not checked for validity. If the VM does not recognize the option, the option is ignored. These options can therefore be used across different VM versions without ensuring a particular level of the VM.

**-XX:allowvmshutdown**

This option is provided as a workaround for customer applications that cannot shut down cleanly, as described in APAR IZ59734.

-XX:allowvmshutdown:[false|true]

Customers who need this workaround should use -XX:allowvmshutdown:false. The default option is -XX:allowvmshutdown:true.

**-XX:codecachetotal**

Use this option to set the maximum size limit for the JIT code cache. This option also affects the size of the JIT data cache.

-XX:codecachetotal=<size>

This option is an alias for the "-Xcodecachetotal" on page 574 option.

**-XX:[+|-]EnableCPUMonitor**

This option relates to the information about the CPU usage of thread categories that is available with the com.ibm.lang.management.JvmCpuMonitorMXBean application programming interface. CPU monitoring is enabled by default, and can be disabled by the command line option -XX:-EnableCPUMonitor. This option might not be supported in subsequent releases.

-XX:[+|-]EnableCPUMonitor

The -XX:+EnableCPUMonitor option enables CPU monitoring, which allows a JMX bean to track CPU usage on a per thread basis and attributes the usage against different categories. For more information, see Interface JvmCpuMonitorMXBean.

The -XX:-EnableCPUMonitor option turns off CPU monitoring.

Related reference:

“-XX:[+|-]ReduceCPUMonitorOverhead” on page 570

This option relates to the information about the CPU usage of thread categories that is available with the com.ibm.lang.management.JvmCpuMonitorMXBean application programming interface. This option affects the way that the JVM records the amount of CPU usage of non-Garbage Collection (GC) threads that do work on behalf of GC.

**-XX:MaxDirectMemorySize**

This option sets a limit on the amount of memory that can be reserved for all Direct Byte Buffers.
-XX:MaxDirectMemorySize=<size>

Where <size> is the limit on memory that can be reserved for all Direct Byte Buffers. If a value is set for this option, the sum of all Direct Byte Buffer sizes cannot exceed the limit. After the limit is reached, a new Direct Byte Buffer can be allocated only when enough old buffers are freed to provide enough space to allocate the new buffer.

By default, the JVM does not set a limit on how much memory is reserved for Direct Byte Buffers. A soft limit of 64 MB is set, which the JVM automatically expands in 32 MB chunks, as required.

-XX:+PackedObject

The -XX:+PackedObject option enables packed object support.

-XX:[+|)[-]PackedObject (Technology preview only)

By default, packed object support is not enabled. For more information, see “Packed object evaluation technology” on page 16.

-XX:+PageAlignDirectMemory

This option affects the alignment of direct byte buffer allocation.

-XX:[+|][-]PageAlignDirectMemory

The -XX:+PageAlignDirectMemory option causes the method java.nio.ByteBuffer.allocateDirect(int) to allocate a direct byte buffer that is aligned on a page boundary. This option sets the sun.nio.PageAlignDirectMemory system property to true.

The -XX:-PageAlignDirectMemory option does not enforce the alignment of the direct byte buffer on a page boundary. This option is the default, which changes previous behavior. Any applications that rely on the alignment, which occurred in Java 6, can revert to this behavior with -XX:+PageAlignDirectMemory.

-XX:[+|-]ReduceCPUMonitorOverhead

This option relates to the information about the CPU usage of thread categories that is available with the com.ibm.lang.management.JvmCpuMonitorMXBean application programming interface. This option affects the way that the JVM records the amount of CPU usage of non-Garbage Collection (GC) threads that do work on behalf of GC.

Most GC policies require non-GC threads to do some GC housekeeping work in proportion to the amount of memory allocation that they do. Ideally the exact amount of CPU time that the thread spends doing this housekeeping work should be accounted for in the GC thread category. However there is an overhead that is associated with maintaining the CPU usage data in the correct thread category.

-XX:+ReduceCPUMonitorOverhead

When this option is set, the JVM does not maintain information on the amount of CPU usage that non-GC threads spend in doing work on behalf of GC. This option is the default setting.

-XX:-ReduceCPUMonitorOverhead

When this option is set, the JVM monitors the amount of GC work that a non-GC thread does and accounts for it in the GC category. This information is made available in the com.ibm.lang.management.JvmCpuMonitorMXBean. Setting this option results in a small increase in application startup time, which varies according to platform.

Related reference:
This option relates to the information about the CPU usage of thread categories that is available with the com.ibm.lang.management.JvmCpuMonitorMXBean application programming interface. CPU monitoring is enabled by default, and can be disabled by the command line option -XX:-EnableCPUMonitor. This option might not be supported in subsequent releases.

-XXsetHWPrefetch
This option enables or disables hardware prefetch.

-XXsetHWPrefetch:[none|os-default] (AIX only)
The -XXsetHWPrefetch:none option disables hardware prefetch. Hardware prefetch can improve the performance of applications by prefetching memory, however because of the workload characteristics of many Java applications, prefetching often has an adverse effect on performance.

You can disable hardware prefetch on AIX by issuing the command dscrct1 -n -s 1. However, this command disables hardware prefetch for all processes, and for all future processes, which might not be desirable in a mixed workload environment. Instead, you can use the -XXsetHWPrefetch:none option to disable hardware prefetch for individual JVMs.

-XXsetHWPrefetch:none is the default setting. To override this setting with the default value for the operating system, specify -XXsetHWPrefetch:os-default. Use this option only for applications that can obtain a performance gain from hardware prefetch.

-XX:ShareClassesEnableBCI
This option is equivalent to -Xshareclasses:enableBCI.

-XX:ShareClassesEnableBCI
-XX:ShareClassesEnableBCI can be specified for any version of the IBM J9 virtual machine, but is ignored by JVMs that are earlier than the IBM J9 2.6 virtual machine. If BCI support is enabled with this option, you can turn off BCI support with -Xshareclasses:disableBCI.

For more information about -Xshareclasses:enableBCI and -Xshareclasses:disableBCI, see "JVM command-line options" on page 549.

-XX:ShareClassesDisableBCI
The option -Xshareclasses:enableBCI is set by default. You can turn off this option by specifying -XX:ShareClassesDisableBCI when you start Java.

-XX:ShareClassesDisableBCI
This option is equivalent to -Xshareclasses:disableBCI.

Related reference:
"-Xshareclasses" on page 560
Enables class sharing. This option can take a number of suboptions, some of which are cache utilities.

-XX:-StackTraceInThrowable
This option removes stack traces from exceptions.

-XX:-StackTraceInThrowable
By default, stack traces are available in exceptions. Including a stack trace in exceptions requires walking the stack and that can affect performance. Removing stack traces from exceptions can improve performance but can also make problems harder to debug.
When this option is enabled, Throwable.getStackTrace() returns an empty array and the stack trace is displayed when an uncaught exception occurs. Thread.getStackTrace() and Thread.getAllStackTraces() are not affected by this option.

**-XX:[+|-]UseCompressedOops (64-bit only)**

This option enables or disables compressed references in 64-bit JVMs, and is provided to help when porting applications from the Oracle JVM to the IBM JVM. This option might not be supported in subsequent releases.

**-XX:[+|-]UseCompressedOops**

The -XX:+UseCompressedOops option enables compressed references in 64-bit JVMs. The -XX:+UseCompressedOops option is similar to specifying -Xcompressedrefs, which is detailed in the topic "JVM command-line options" on page 549.

The -XX:-UseCompressedOops option prevents the use of compressed references in 64-bit JVMs.

**-XX:[+|-]VerboseVerification**

You can use this option to control the output of verbose diagnostic data that relates to verification.

**-XX:+VerboseVerification**

This option enables the output of verbose diagnostic data to stderr that is generated during verification from the class file StackMapTable attribute. The data provides extra contextual information about bytecode verification, which helps diagnose bytecode or stackmap deficiencies in the field.

Class files that have StackMapTable attributes (that is, class files that conform to version 50.0 or later of the class file format specification), are introduced in Java V6. Class files with StackMapTable attributes are marked as new format in the verbose output, as shown in the example. Class files without the StackMapTable attributes are marked as old format. The StackMapTable diagnostic information is available only to classes verified with the new format.

**-XX:-VerboseVerification**

Use this option to turn off verbose output. This option is the default.

Here is an example of StackMapTable diagnostic output:

```bash
Verifying class java.example.ibm.com with new format
Verifying method java.example.ibm.com.foo(ljava/lang/String;Ljava/lang/Class;[Ljava/lang/String;Ljava/io/PrintStream;)I
StackMapTable: frame_count = 3
 table = {
  bci: 037
  flags: { }
  locals: { 'java/lang/String', 'java/lang/Class', '[Ljava/lang/String;', 'java/io/PrintStream', 'java/lang/Class' } 
  stack: { 'java/lang/ThreadDeath' }
  bci: 042
  flags: { }
  locals: { 'java/lang/String', 'java/lang/Class', '[Ljava/lang/String;', 'java/io/PrintStream', 'java/lang/Class' } 
  stack: { 'java/lang/Throwable' }
  bci: 079
  flags: { }
  locals: { 'java/lang/String', 'java/lang/Class', '[Ljava/lang/String;', 'java/io/PrintStream', 'java/lang/Class', 'java/lang/Throwable' } 
  stack: { }
}
End class verification for: java.example.ibm.com
```
**-XX:[+|-]VMLockClassLoader**

This option affects synchronization on class loaders that are not parallel-capable class loaders, during class loading.

**-XX:[+|-]VMLockClassLoader**

The option, `-XX:+VMLockClassLoader`, causes the JVM to force synchronization on a class loader that is not a parallel capable class loader during class loading. This action occurs even if the `loadClass()` method for that class loader is not synchronized. For information about parallel capable class loaders, see `java.lang.ClassLoader.registerAsParallelCapable()` in Java 7 or later. Note that this option might cause a deadlock if class loaders use non-hierarchical delegation. For example, setting the system property `osgi.classloader.lock=classname` with Equinox is known to cause a deadlock. This is the default option.

When specifying the `-XX:-VMLockClassLoader` option, the JVM does not force synchronization on a class loader during class loading. The class loader still conforms to class library synchronization, such as a synchronized `loadClass()` method.

## JIT and AOT command-line options

Use these JIT and AOT compiler command-line options to control code compilation.

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

For more information about JIT and AOT, see “JIT and AOT problem determination” on page 430.

**-Xaot**

Use this option to control the behavior of the AOT compiler.

**-Xaot[<parameter>=<value>, ...]**

With no parameters, enables the AOT compiler. The AOT compiler is enabled by default but is not active unless shared classes are enabled. Using this option on its own has no effect. The following parameters are useful:

- `count=<n>`
  
  Where `<n>` is the number of times a method is called before it is compiled or loaded from an existing shared class cache. For example, setting `count=0` forces the AOT compiler to compile everything on first execution.

- `exclude={<method>}`
  
  Where `<method>` is the Java method you want to exclude when AOT code is compiled or loaded from the shared classes cache. You can use this option if the method causes the program to fail.

- `limitFile=<filename>,<m>,<n>`
  
  Compile or load only the methods listed on lines `<m>` to `<n>` in the specified limit file. Methods not listed in the limit file and methods listed on lines outside the range are not compiled or loaded.

- `loadExclude=<methods>`
  
  Do not load methods beginning with `<methods>`.

- `loadLimit=<methods>`
  
  Load methods beginning with `<methods>` only.
loadLimitFile=(<filename>,<m>,<n>)
Load only the methods listed on lines <m> to <n> in the specified limit file. Methods not listed in the limit file and methods listed on lines outside the range are not loaded.

verbose
Reports information about the AOT and JIT compiler configuration and method compilation.

-Xcodecache
This option is used to tune performance.

-Xcodecache=<size>
This option sets the size of each block of memory that is allocated to store the native code of compiled Java methods. By default, this size is selected internally according to the processor architecture and the capability of your system. The maximum value a user can specify is 32 MB. If you set a value larger than 32 MB, the JIT ignores the input and sets the value to 32 MB.

Note: The JIT compiler might allocate more than one code cache for an application. Use the -Xcodecachetotal option to set the maximum amount of memory that is used by all code caches.

-Xcodecachetotal
Use this option to set the maximum size limit for the JIT code cache. This option also affects the size of the JIT data cache.

-Xcodecachetotal=<size>
See “JIT and AOT command-line options” on page 573 for more information about the <size> parameter.

By default, the total size of the JIT code cache is determined by your operating system, architecture, and the version of the IBM SDK that you are using. Long-running, complex, server-type applications can fill the JIT code cache, which can cause performance problems because not all of the important methods can be JIT-compiled. Use the -Xcodecachetotal option to increase the maximum code cache size beyond the default setting, to a setting that suits your application.

The value that you specify is rounded up to a multiple of the code cache block size, as specified by the -Xcodecache option. If you specify a value for the -Xcodecachetotal option that is smaller than the default setting, that value is ignored.

When you use this option, the maximum size limit for the JIT data cache, which holds metadata about compiled methods, is increased proportionally to support the additional JIT compilations.

The maximum size limits, for both the JIT code and data caches, that are in use by the JVM are shown in Javadump output. Look for lines that begin with 1STSEGLIMIT. Use this information together with verbose JIT tracing to determine suitable values for this option on your system. For example Javadump output, see “Storage Management (MEMINFO)” on page 348.

Related reference:
“-Xjit” on page 575
Use the JIT compiler command line option to produce verbose JIT trace output.

Related information:
Javadump produces files that contain diagnostic information that is related to the JVM and a Java application that is captured at a point during execution. For example, the information can be about the operating system, the application environment, threads, stacks, locks, and memory.

-XcompilationThreads
Use this option to specify the number of compilation threads that are used by the JIT compiler.

-XcompilationThreads<number of threads>
The number of threads must be in the range 1 - 4, inclusive. Any other value prevents the Java VM 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 uses the following pattern:

<specified_filename>.<date>.<time>.<pid>

The first compilation thread has ID 0. Log files generated by the second and subsequent compilation threads append the ID of the corresponding compilation thread as a suffix to the log file name. The pattern for these log file names is as follows:

<specified_filename>.<date>.<time>.<pid>.<compThreadID>

For example, the second compilation thread has ID 1. The result is that the corresponding log file name has the form:

<specified_filename>.<date>.<time>.<pid>.1

-Xint
This option makes the JVM use the Interpreter only, disabling the Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilers.

-Xint
By default, the JIT compiler is enabled. By default, the AOT compiler is enabled, but is not used by the JVM unless shared classes are also enabled.

-Xjit
Use this option to control the behavior of the JIT compiler.

-Xjit[::<option>=<value>, ...]
The JIT compiler is enabled by default. Therefore, specifying -Xjit with no options, has no effect. These options can be used to modify behavior:

count=<n>
Where <n> is the number of times a method is called before it is compiled. For example, setting count=0 forces the JIT compiler to compile everything on first execution.

exclude={<method>}
Excludes the specified method from compilation.

limitFile=(<filename>, <m>, <n>)
Compile only the methods that are listed on lines <m> to <n> in the specified limit file. Methods that are not listed in the limit file and methods that are listed on lines outside the range are not compiled.
optlevel=[ noOpt | cold | warm | hot | veryHot | scorching ]
Forces the JIT compiler to compile all methods at a specific optimization
level. Specifying optlevel might have an unexpected effect on
performance, including reduced overall performance.

verbose={compileStart|compileEnd}
Reports information about the JIT and AOT compiler configuration and
method compilation.
The ={compileStart|compileEnd} option reports when the JIT starts to
compile a method, and when it ends.

vlog={filename>
Sends verbose output to a file. If you do not specify this parameter, the
output is sent to the standard error output stream (STDERR).

Related tasks:
"Diagnosing a JIT or AOT problem” on page 430
Occasionally, valid bytecodes might compile into invalid native code, causing the
Java program to fail. By determining whether the JIT or AOT compiler is faulty
and, if so, where it is faulty, you can provide valuable help to the Java service team.

-Xnoaot
This option turns off the AOT compiler and disables the use of AOT-compiled
code.

-Xnoaot
By default, the AOT compiler is enabled but is active only when shared classes
are also enabled. Using this option does not affect the JIT compiler.

-Xnojit
This option turns off the JIT compiler.

-Xnojit
By default, the JIT compiler is enabled. This option does not affect the AOT
compiler.

-Xquickstart
This option causes the JIT compiler to run with a subset of optimizations.

-Xquickstart
The effect is faster compilation times that improve startup time, but longer
running applications might run slower. When the AOT compiler is active (both
shared classes and AOT compilation enabled), -Xquickstart causes all methods
to be AOT compiled. The AOT compilation improves the startup time of
subsequent runs, but might reduce performance for longer running
applications. -Xquickstart can degrade performance if it is used with
long-running applications that contain hot methods. The implementation of
-Xquickstart is subject to change in future releases. By default, -Xquickstart
is disabled.

Another way to specify a behavior identical to -Xquickstart is to use the
-client option. These two options can be used interchangeably on the
command line.

-XsamplingExpirationTime
Use this option to disable JIT sampling after a specified amount of time.
-XsamplingExpirationTime<time>
Disables the JIT sampling thread after <time> seconds. When the JIT sampling thread is disabled, no processor cycles are used by an idle JVM.

-Xscmaxaot
When you create a shared classes cache, you can use this option to apply a maximum number of bytes in the class cache that can be used for AOT data.

-Xscmaxaot<size>
This option is useful if you want a certain amount of cache space guaranteed for non-AOT data. If this option is not specified, by default 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
When you create a shared classes cache, you can use this option to apply a maximum number of bytes in the class cache that can be used for JIT data.

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

-Xscminaot
When you create a shared classes cache, you can use this option to apply a minimum number of bytes in the class cache to reserve for AOT data.

-Xscminaot<size>
If this option is not specified, no space is reserved for AOT data. However, AOT data is still written to the cache until the cache is full or the -Xscmaxaot limit is reached. The value of this option must not exceed the value of -Xscmx or -Xscmaxaot. The value of -Xscminaot must always be considerably less than the total cache size, because AOT data can be created only for cached classes. If the value of -Xscminaot equals the value of -Xscmx, no class data or AOT data can be stored.

-Xscminjitdata
When you create a shared classes cache, you can use this option to apply a minimum number of bytes in the class cache to reserve for JIT data.

-Xscminjitdata<x>
If this option is not specified, no space is reserved for JIT data, although JIT data is still written to the cache until the cache is full or the -Xscmaxjit limit is reached. The value of this option must not exceed the value of -Xscmx or -Xscmaxjitdata. The value of -Xscminjitdata must always be considerably less than the total cache size, because JIT data can be created only for cached classes. If the value of -Xscminjitdata equals the value of -Xscmx, no class data or JIT data can be stored.

Garbage Collector command-line options
Use these Garbage Collector command-line options to control garbage collection.

You might need to read “Memory management” on page 69 to understand some of the references that are given here.
The `-verbose:gc` option detailed in “Verbose garbage collection logging” on page 437 is the main diagnostic aid that is available for runtime analysis of the Garbage Collector. However, additional command-line options are available that affect the behavior of the Garbage Collector and might aid diagnostic data collection.

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

For options that take a `<percentage>` parameter, use a number from 0 to 1, for example, 50% is 0.5.

**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 84. For information about when to use this policy, see “When to use the Balanced garbage collection policy” on page 89.

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>
- Xmns<size>
- Xmnx<size>
- Xms<size>
- Xmx<size>
- Xnogc
- Xnocompactexplicitgc
- Xnocompactgc
- Xnocompactexplicitgc
- Xnuma:none
- Xsoftmx<size>
- Xsoftrefthreshold<number>
- Xverbosegclog[:<file> [, <X>,<Y>]]

A detailed description of these command-line options can be found in “Garbage Collector command-line options” on page 577.

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
- Xloa
- Xloainitial<percentage>
- Xloamaximum<percentage>
- Xloaminimum<percentage>
- Xmo<size>
- Xmoi<size>
- Xmos<size>
- Xmrgc
- Xmrx<size>
- Xnoloa
- Xnopartialcompactgc (deprecated)
- Xpartialcompactgc (deprecated)

A detailed description of these command-line options can be found in “Garbage Collector command-line options” on page 577.
**Metronome Garbage Collector policy options**
The definitions of the Metronome Garbage Collector options.

- `-Xgc:synchronousGCOnOOM` | `-Xgc:nosynchronousGCOnOOM`
  One occasion when garbage collection occurs is when the heap runs out of memory. If there is no more free space in the heap, using `-Xgc:synchronousGCOnOOM` stops your application while garbage collection removes unused objects. If free space runs out again, consider decreasing the target utilization to allow garbage collection more time to complete. Setting `-Xgc:nosynchronousGCOnOOM` implies that when heap memory is full your application stops and issues an out-of-memory message. The default is `-Xgc:synchronousGCOnOOM`.

- `-Xnoclassgc`
  Disables class garbage collection. This option switches off garbage collection of storage associated with Java classes that are no longer being used by the JVM. The default behavior is `-Xnoclassgc`.

- `-Xgc:targetPauseTime=N`
  Sets the garbage collection pause time, where `N` is the time in milliseconds. When this option is specified, the GC operates with pauses that do not exceed the value specified. If this option is not specified the default pause time is set to 3 milliseconds. For example, running with `-Xgc:targetPauseTime=20` causes the GC to pause for no longer than 20 milliseconds during GC operations.

- `-Xgc:targetUtilization=N`
  Sets the application utilization to `N%`; the Garbage Collector attempts to use at most (100-N)% of each time interval. Reasonable values are in the range of 50-80%. Applications with low allocation rates might be able to run at 90%. The default is 70%.

  This example shows the maximum size of the heap memory is 30 MB. The garbage collector attempts to use 25% of each time interval because the target utilization for the application is 75%.

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

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

- `-Xgc:verboseGCCycleTime=N`
  `N` is the time in milliseconds that the summary information should be dumped.

  **Note:** The cycle time does not mean that the summary information is dumped precisely at that time, but when the last garbage collection event that meets this time criterion passes.

- `-Xmx<size>`
  Specifies the Java heap size. Unlike other garbage collection strategies, the real-time Metronome GC does not support heap expansion. There is not an initial or maximum heap size option. You can specify only the maximum heap size.

- `-Xalwaysclassgc`
  Always perform dynamic class unloading checks during global collection.
-Xalwaysclassgc
   The default behavior is as defined by -Xclassgc.

-XXclassgc
   Enables dynamic unloading of classes by the JVM. Garbage collection of class objects occurs only on class loader changes.
   Dynamic unloading is the default behavior. To disable dynamic class unloading, use the -Xnoclassgc option.

-XXcompactexplicitgc
   Enables full compaction each time System.gc() is called.

-XXcompactexplicitgc
   Enables full compaction each time System.gc() is called.

-XXcompactgc
   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.

-XXconcurrentbackground
   Specifies the number of low-priority background threads attached to assist the mutator threads in concurrent mark.
   -XXconcurrentbackground<number>
      The default is 0 for Linux on IBM Z and 1 on all other platforms.

-XXconcurrentlevel
   Specifies the allocation "tax" rate.
   -XXconcurrentlevel<number>
      This option indicates the ratio between the amount of heap allocated and the amount of heap marked. The default is 8.

-XXconcurrentslack
   Attempts to keep the specified amount of the heap space free in concurrent collectors by starting the concurrent operations earlier.
   -XXconcurrentslack<size>
      Using this option can sometimes alleviate pause time problems in concurrent collectors at the cost of longer concurrent cycles, affecting total throughput. The default value is 0, which is optimal for most applications.

-XXconmter
   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.
   -XXconmeter:<soa | loa | dynamic>
      Using -XXconmeter:soa (the default) applies the allocation tax to allocations from the small object area (SOA). Using -XXconmeter:loa applies the allocation tax to allocations from the large object area (LOA). If -XXconmeter:dyanmic is specified, the collector dynamically determines which area to meter based on which area is exhausted first, whether it is the SOA or the LOA.
**-Xdisableexcessivegc**
Disables the throwing of an OutOfMemory exception if excessive time is spent in the GC.

**-Xdisableexcessivegc**
Disables the throwing of an OutOfMemory exception if excessive time is spent in the GC.

**-Xdisableexplicitgc**
Disables System.gc() calls.

**-Xdisableexplicitgc**
Many applications still make an excessive number of explicit calls to System.gc() to request garbage collection. In many cases, these calls degrade performance through premature garbage collection and compactions. However, you cannot always remove the calls from the application.

The **-Xdisableexplicitgc** parameter allows the JVM to ignore these garbage collection suggestions. Typically, system administrators use this parameter in applications that show some benefit from its use.

By default, calls to System.gc() trigger a garbage collection.

**-Xdisablestringconstantgc**
Prevents strings in the string intern table from being collected.

**-Xdisablestringconstantgc**
Prevents strings in the string intern table from being collected.

**-Xenableexcessivegc**
If excessive time is spent in the GC, the option returns null for an allocate request and thus causes an OutOfMemory exception to be thrown.

**-Xenableexcessivegc**
The OutOfMemory exception is thrown only when the heap has been fully expanded and the percentage of application run time that is not spent in garbage collection is at least 95%. This percentage is the default value that triggers an excessive GC event. You can control this value with the **-Xgc:excessiveGCratio** option. For more information, see "-Xgc."

**-Xenablestringconstantgc**
Enables strings from the string intern table to be collected.

**-Xenablestringconstantgc**
This option is on by default.

**-Xgc**
Options that change the behavior of the Garbage Collector (GC).

**-Xgc:<dnssExpectedTimeRatioMaximum | dnssExpectedTimeRatioMinimum | excessiveGCratio | minContractPercent | maxContractPercent | overrideHiresTimerCheck | preferredHeapBase | scvNoAdaptiveTenure | scvTenureAge | verboseFormat>**

**dnssExpectedTimeRatioMaximum=value**
The maximum amount of time spent on garbage collection of the nursery area, expressed as a percentage of the overall time for the last three GC intervals. The default is 5%.
dnssExpectedTimeRatioMinimum=value
The minimum amount of time spent on garbage collection of the
nursery area, expressed as a percentage of the overall time for the last
three GC intervals. The default is 1%.

excessiveGCratio=value
Where value is a percentage of total application run time that is not
spent in GC. The default value is 95, which means that anything over
5% of total application run time spent on GC is deemed excessive. This
option can be used only when -Xenableexcessivegc is set. For more
information, see "-Xenableexcessivegc" on page 582.

minContractPercent=<n>
The minimum percentage of the heap that can be contracted at any
given time.

maxContractPercent=<n>
The maximum percentage of the heap that can be contracted at any
given time. For example, -Xgc:maxContractPercent=20 causes the heap
to contract by as much as 20%.

overrideHiresTimerCheck
When the JVM starts, the GC checks that the operating system can
meet the timer resolution requirements for the requested target pause
time. Typically, this check correctly identifies operating systems that
can deliver adequate time resolution. However, in some cases the
operating system provides a more conservative answer than strictly
necessary for GC pause time management, which prevents startup.
Specifying the -Xgc:overrideHiresTimerCheck option causes the GC to
ignore the answer returned by the operating system. The JVM starts,
but GC pause time management remains subject to operating system
performance, which might not provide adequate timer resolution.

Note: Use this option with caution, and only when you are unable to
use a supported operating system.

preferredHeapBase=<address>
Where, <address> is the base memory address for the heap. Use this
option with the -Xcompressedrefs option to allocate the heap you
specify with the -Xmx option, in a memory range of your choice. If
-Xcompressedrefs is not specified, this option has no effect. In the
following example, the heap is located at the 4 GB mark, leaving the
lowest 4 GB of address space for use by other processes.
-Xgc:preferredHeapBase=0x100000000

If the heap cannot be allocated in a contiguous block at the
preferredHeapBase address you specified, an error occurs detailing a
Garbage Collection (GC) allocation failure startup. When the
preferredHeapBase option is used with the -X1p option, the
preferredHeapBase address must be a multiple of the large page size. If
you specify an inaccurate heap base address, the heap is allocated with
the default page size.

scvNoAdaptiveTenure
This option turns off the adaptive tenure age in the generational
concurrent GC policy. The initial age that is set is maintained
throughout the run time of the Java virtual machine. See scvTenureAge.
scvTenureAge=<n>
This option sets the initial scavenger tenure age in the generational concurrent GC policy. The range is 1 - 14 and the default value is 10. For more information, see “Tenure age” on page 84.

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 437.
- deprecated: The verbose garbage collection format available in Version 6 and earlier releases of the SDK. For more information, see the v6 Diagnostic Guide.

-Xgcpolicy
Controls the behavior of the Garbage Collector.

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

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

balanced
The balanced policy uses mark, sweep, compact and generational style garbage collection. The concurrent mark phase is disabled; concurrent garbage collection technology is used, but not in the way that concurrent mark is implemented for other policies. The balanced policy uses a region-based layout for the Java heap. These regions are individually managed to reduce the maximum pause time on large heaps and increase the efficiency of garbage collection. The policy tries to avoid global collections by matching object allocation and survival rates. If you have problems with application pause times that are caused by global garbage collections, particularly compactions, this policy might improve application performance. If you are using large systems that have Non-Uniform Memory Architecture (NUMA) characteristics (x86 and POWER platforms only), the balanced policy might further improve application throughput. For more information about this policy, including when to use it, see “Balanced Garbage Collection policy” on page 84.

metronome
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. The metronome policy is supported on specific hardware and operating system configurations. For more information, see “Using the Metronome Garbage Collector” on page 218.

optavgpause
The “optimize for pause time” (optavgpause) policy uses concurrent mark and concurrent sweep phases. Pause times are shorter than with optthruput, but application throughput is reduced because some garbage collection work is taking place while the application is running. Consider using this policy if you have a large heap size (available on 64-bit...
platforms), because this policy limits the effect of increasing heap size on the length of the garbage collection pause. However, if your application uses many short-lived objects, the gencon policy might produce better performance.

**subpool**

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

**optthruput**

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

**-Xgcthreads**

Sets the number of threads that the Garbage Collector uses for parallel operations.

**-Xgcthreads<number>**

The total number of GC threads is composed of one application thread with the remainder being dedicated GC threads. By default, the number is set to \( n-1 \), where \( n \) is the number of reported CPUs, up to a maximum of 64. Where SMT or hyperthreading is in place, the number of reported CPUs is larger than the number of physical CPUs. Likewise, where virtualization is in place, the number of reported CPUs is the number of virtual CPUs assigned to the operating system. To set it to a different number, for example 4, use **-Xgcthreads4**. The minimum valid value is 1, which disables parallel operations, at the cost of performance. No advantage is gained if you increase the number of threads to more than the default setting.

On systems running multiple JVMs or in LPAR environments where multiple JVMs can share the same physical CPUs, you might want to restrict the number of GC threads used by each JVM. The restriction helps prevent the total number of parallel operation GC threads for all JVMs exceeding the number of physical CPUs present, when multiple JVMs perform garbage collection at the same time.

**-Xgcworkpackets**

Specifies the total number of work packets available in the global collector.

**-Xgcworkpackets<number>**

If you do not specify a value, the collector allocates a number of packets based on the maximum heap size.

**-Xloa**

This option enables the large object area (LOA).

**-Xloa**

By default, allocations are made in the small object area (SOA). If there is no room in the SOA, and an object is larger than 64KB, the object is allocated in the LOA.

By default, the LOA is enabled for all GC policies.

**-Xloainitilal**

Specifies the initial percentage (between 0 and 0.95) of the current tenure space allocated to the large object area (LOA).
**-Xloainitial <percentage>**
The default value is 0.05, which is 5%.

**-Xloamaximum**
Specifies the maximum percentage (between 0 and 0.95) of the current tenure space allocated to the large object area (LOA).

**-Xloamaximum <percentage>**
The default value is 0.5, which is 50%.

**-Xloaminimum**
Specifies the minimum percentage (between 0 and 0.95) of the current tenure space allocated to the large object area (LOA).

**-Xloaminimum <percentage>**
The LOA does not shrink to less than this value. The default value is 0, which is 0%.

**-Xmaxe**
Sets the maximum amount by which the garbage collector expands the heap.

**-Xmaxe <size>**
Typically, the garbage collector expands the heap when the amount of free space falls to less than 30% (or by the amount specified using `-Xminf`), by the amount required to restore the free space to 30%. The `-Xmaxe` option limits the expansion to the specified value; for example `-Xmaxe10M` limits the expansion to 10 MB. By default, there is no maximum expansion size.

**-Xmaxf**
Specifies the maximum percentage of heap that must be free after a global garbage collection cycle.

**-Xmaxf <percentage>**
If the free space exceeds this amount, the JVM tries to shrink the heap. The default value is 0.6 (60%).

**-Xmaxt**
Specifies the maximum percentage of time to be spent in Garbage Collection (GC).

**-Xmaxt <percentage>**
The maximum time that an application spends in the garbage collection process as a percentage of the overall running time that included the last three GC runs. If the percentage of time exceeds this value, the JVM tries to expand the heap. The default value is 13%.

**Note:** This option applies only to GC policies that include stop-the-world (STW) operations, such as `-Xgcpolicy:optthruput`. This option is ignored by the default policy `-Xgcpolicy:gencon`.

**Related reference:**

**"-Xmint" on page 587**
Specifies the minimum percentage of time to spend in Garbage Collection (GC).

**-Xmca**
Sets the expansion step for the memory allocated to store the RAM portion of loaded classes.

**-Xmca <size>**
Each time more memory is required to store classes in RAM, the allocated memory is increased by this amount. By default, the expansion step is 32 KB.
Use the `-verbose:sizes` option to determine the value that the VM is using. If the expansion step size you choose is too large, `OutOfMemoryError` is reported. The exact value of a “too large” expansion step size varies according to the platform and the specific machine configuration.

**-Xmcrs**

Sets an initial size for an area in memory that is reserved for compressed references within the lowest 4 GB memory area.

Native memory `OutOfMemoryError` exceptions might occur when using compressed references if the lowest 4 GB of address space becomes full, particularly when loading classes, starting threads, or using monitors. This option secures space for any native classes, monitors, and threads that are used by compressed references.

**-Xmcrs**<mem_size>

Where `<mem_size>` is the initial size. You can use the `-verbose:sizes` option to find out the value that is being used by the VM. If you are not using compressed references and this option is set, the option is ignored and the output of `-verbose:sizes` shows `-Xmcrs0`.

The following option sets an initial size of 200 MB for the memory area:

`-Xmcrs200M`

**-Xmco**

Sets the expansion step for the memory allocated to store the ROM portion of loaded classes.

**-Xmco**<size>

Each time more memory is required to store classes in ROM, the allocated memory is increased by this amount. By default, the expansion step is 128 KB. Use the `-verbose:sizes` option to determine the value that the VM is using. If the expansion step size you choose is too large, `OutOfMemoryError` is reported. The exact value of a “too large” expansion step size varies according to the platform and the specific machine configuration.

**-Xmine**

Sets the minimum amount by which the Garbage Collector expands the heap.

**-Xmine**<size>

Typically, the garbage collector expands the heap by the amount required to restore the free space to 30% (or the amount specified using `-Xmminf`). 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.

**-Xmminf**

Specifies the minimum percentage of heap to remain free after a global garbage collection cycle.

**-Xmminf**<percentage>

If the free space falls to less than this amount, the JVM attempts to expand the heap. The default value is 30%.

**-Xmint**

Specifies the minimum percentage of time to spend in Garbage Collection (GC).

**-Xmint**<percentage>

The minimum time that an application spends in the garbage collection process.
as a percentage of the overall running time that included the last three GC
runs. If the percentage of time exceeds this value, the JVM tries to expand the
heap. The default value is 5%. If the percentage of time drops to less than this
value, the JVM tries to shrink the heap.

Note: This option applies only to GC policies that include stop-the-world (STW)
operations, such as -Xgcpolicy:optthruput. This option is ignored by the default
policy -Xgcpolicy:gencon.

Related reference:
“-Xmmax” on page 586
Specifies the maximum percentage of time to be spent in Garbage Collection (GC).

-Xmn
Sets the initial and maximum size of the new area to the specified value when
using -Xgcpolicy:gencon.

-Xmn<size>
Equivalent to setting both -Xmns and -Xmnx. If you set either -Xmns or -Xmnx,
you cannot set -Xmn. If you try to set -Xmn with either -Xmns or -Xmnx, the VM
does not start, returning an error. By default, -Xmn is not set. If the scavenger is
disabled, this option is ignored.

-Xmns
Sets the initial size of the new area to the specified value when using
-Xgcpolicy:gencon.

-Xmns<size>
By default, this option is set to 25% of the value of the -Xms option. This option
returns an error if you try to use it with -Xmn. You can use the -verbose:sizes
option to find out the values that the VM is currently using. If the scavenger is
disabled, this option is ignored.

-Xmnx
Sets the maximum size of the new area to the specified value when using
-Xgcpolicy:gencon.

-Xmnx<size>
By default, this option is set to 25% of the value of the -Xmx option. This option
returns an error if you try to use it with -Xmn. You can use the -verbose:sizes
option to find out the values that the VM is currently using. If the scavenger is
disabled, this option is ignored.

-Xmo
Sets the initial and maximum size of the old (tenured) heap to the specified value
when using -Xgcpolicy:gencon.

-Xmo<size>
Equivalent to setting both -Xmos and -Xmox. If you set either -Xmos or -Xmox,
you cannot set -Xmo. If you try to set -Xmo with either -Xmos or -Xmox, the VM
does not start, returning an error. By default, -Xmo is not set.

-Xmoi
Sets the amount the Java heap is incremented when using -Xgcpolicy:gencon.

-Xmoi<size>
If set to zero, no expansion is allowed. By default, the increment size is
calculated on the expansion size, set by -Xmmin and -Xminf.
-Xmos
Sets the initial size of the old (tenure) heap to the specified value when using -Xgcpolicy:gencon.

-Xmos<size>
By default, this option is set to 75% of the value of the -Xms option. This option returns an error if you try to use it with -Xmo. You can use the -verbose:sizes option to find out the values that the VM is currently using.

-Xmox
Sets the maximum size of the old (tenure) heap to the specified value when using -Xgcpolicy:gencon.

-Xmox<size>
By default, this option is set to the same value as the -Xmx option. This option returns an error if you try to use it with -Xmo. You can use the -verbose:sizes option to find out the values that the VM is currently using.

-Xmr
Sets the size of the Garbage Collection "remembered set".

-Xmr<size>
The Garbage Collection "remembered set" is a list of objects in the old (tenured) heap that have references to objects in the new area. By default, this option is set to 16 K.

-XmrX
Sets the remembered maximum size setting.

-XmrX<size>
Sets the remembered maximum size setting.

-Xms
Sets the initial Java heap size.

-Xms<size>
size can be specified in megabytes (m) or gigabytes (g). For example: -Xms2g sets an initial Java heap size of 2GB. The minimum size is 1 MB.

You can also use the -Xmo option.

If the scavenger is enabled, -Xms >= -Xmn + -Xmo.

If the scavenger is disabled, -Xms >= -Xmo.

Note: The -Xmo option is not supported by the balanced garbage collection policy.

-Xmx
Sets the maximum memory size for the application (-Xmx >= -Xms).

-Xmx<size>
size can be specified in megabytes (m) or gigabytes (g). For example: -Xmx2g sets a maximum heap size of 2GB.

For information about default values, see "Default settings for the JVM" on page 603.

If you are allocating the Java heap with large pages, read the information provided for the "-Xlp" on page 555 option and "More effective heap usage using compressed references" on page 216.
Examples of the use of `-Xms` and `-Xmx`:

- `-Xms2m -Xmx64m`
  Heap starts at 2 MB and grows to a maximum of 64 MB.

- `-Xms100m -Xmx100m`
  Heap starts at 100 MB and never grows.

- `-Xms20m -Xmx1024m`
  Heap starts at 20 MB and grows to a maximum of 1 GB.

- `-Xms50m`
  Heap starts at 50 MB and grows to the default maximum.

- `-Xmx256m`
  Heap starts at default initial value and grows to a maximum of 256 MB.

If you exceed the limit set by the `-Xmx` option, the JVM generates an OutofMemoryError.

**-Xnoclassgc**
Disables class garbage collection.

**-Xnoclassgc**
This option switches off garbage collection of storage associated with Java technology 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 support team. The reason is the option can cause unlimited native memory growth, leading to out-of-memory errors.

**-Xnocompactexplicitgc**
Disables compaction on System.gc() calls.

**-Xnocompactexplicitgc**
Compaction takes place on global garbage collections if you specify `-Xcompactgc` or if compaction triggers are met. By default, compaction is enabled on calls to System.gc().

**-Xnocompactgc**
Disables compaction on all garbage collections (system or global).

**-Xnocompactgc**
By default, compaction is enabled.

**-Xnoloa**
Prevents allocation of a large object area; all objects are allocated in the SOA.

**-Xnoloa**
See also `-Xloa`.

**-Xnuma:none**
Use this option to turn off Non-Uniform Memory Architecture (NUMA) awareness when using the balanced garbage collection policy.

**-Xnuma:none**
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**
This option sets a "soft" maximum limit for the initial size of the Java heap.
Use the -Xmx option to set a "hard" limit for the maximum size of the heap. By default, -Xsoftmx is set to the same value as -Xmx. The value of -Xms must be less than, or equal to, the value of -Xsoftmx. See the introduction to this topic for more information about specifying <size> parameters.

You can set this option on the command line, then modify it at run time by using the MemoryMXBean.setMaxHeapSize() method in the com.ibm.lang.management API. By using this API, Java applications can dynamically monitor and adjust the heap size as required. This function can be useful in virtualized or cloud environments, for example, where the available memory might change dynamically to meet business needs. When you use the API, you must specify the value in bytes, such as 2147483648 instead of 2g.

For example, you might set the initial heap size to 1 GB and the maximum heap size to 8 GB. You might set a smaller value, such as 2 GB, for -Xsoftmx, to limit the heap size that is used initially:

```
-Xms1g -Xsoftmx2g -Xmx8g
```

You can then use the com.ibm.lang.management API from within a Java application to increase the -Xsoftmx value during run time, as load increases. This change allows the application to use more memory than you specified initially.

If you reduce the -Xsoftmx value, the garbage collector attempts to respect the new limit. However, the ability to shrink the heap depends on a number of factors. There is no guarantee that a decrease in the heap size will occur. If or when the heap shrinks to less than the new limit, the heap will not grow beyond that limit.

When the heap shrinks, the garbage collector might release memory. The ability of the operating system to reclaim and use this memory varies based on the capabilities of the operating system.

Notes:

- When using -Xgcpolicy:gencon, -Xsoftmx applies only to the non-nursery portion of the heap. In some cases the heap grows to greater than the -Xsoftmx value because the nursery portion grows, making the heap size exceed the limit that is set. See -Xmn for limiting the nursery size.
- When using -Xgcpolicy:metronome, -Xsoftmx is ignored because the Metronome garbage collector does not support contraction or expansion of the heap.
- There might be little benefit in reducing the -Xsoftmx value when the Java heap is using large pages. Large pages are pinned in memory and are not reclaimed by the operating system, with the exception of 1M pageable pages on z/OS. On certain platforms and processors the JVM starts with large pages enabled by default for the Java heap when the operating system is configured to provide large pages. For more information, see "Configuring large page memory allocation" on page 231. A future version of the Java virtual machine might provide a hint to the operating system when large pages are no longer in use.

-Xsoftrefthreshold

Sets the value used by the garbage collector to determine the number of garbage collections after which a soft reference is cleared if its referent has not been marked.
-Xsoftrefthreshold<number>

The default is 32, meaning that the soft reference is cleared after 32 * (percentage of free heap space) garbage collection cycles where its referent was not marked. For example, if -Xsoftrefthreshold is set to 32, and the heap is 50% free, soft references are cleared after 16 garbage collection cycles.

-Xtgc

Provides garbage collection tracing options.

-Xtgc:<arguments>

<arguments> is a comma-separated list containing one or more of the following arguments:

backtrace

Before a garbage collection, a single line is printed containing the name of the master thread for garbage collection, as well as the value of the osThread slot in the J9VMThread structure.

compaction

Prints extra information showing the relative time spent by threads in the “move” and “fixup” phases of compaction

concurrent

Prints extra information showing the activity of the concurrent mark background thread

dump

Prints a line of output for every free chunk of memory in the system, including "dark matter" (free chunks that are not on the free list for some reason, typically because they are too small). Each line contains the base address and the size in bytes of the chunk. If the chunk is followed in the heap by an object, the size and class name of the object is also printed. This argument has a similar effect to the terse argument.

freelist

Before a garbage collection, prints information about the free list and allocation statistics since the last garbage collection. Prints the number of items on the free list, including "deferred” entries (with the scavenger, the unused space is a deferred free list entry). For TLH and non-TLH allocations, prints the total number of allocations, the average allocation size, and the total number of bytes discarded during allocation. For non-TLH allocations, also included is the average number of entries that were searched before a sufficiently large entry was found.

parallel

Produces statistics on the activity of the parallel threads during the mark and sweep phases of a global garbage collection.

scavenger

Prints extra information after each scavenger collection. A histogram is produced showing the number of instances of each class, and their relative ages, present in the survivor space. The information is obtained by performing a linear walk-through of the space.

terse

Dumps the contents of the entire heap before and after a garbage collection. For each object or free chunk in the heap, a line of trace output is produced. Each line contains the base address, ”a” if it is an allocated object, and ”f” if it is a free chunk, the size of the chunk in bytes, and, if it is an object, its class name.
-Xverbosegclog
Causes -verbose:gc output to be written to a specified file.

-Xverbosegclog[:<file>[,[<X>,<Y>]]]
If the file cannot be found, -verbose:gc tries to create the file, and then continues as normal if it is successful. If it cannot create the file (for example, if an invalid filename is passed into the command), it redirects the output to stderr.

If you specify <X> and <Y> the -verbose:gc output is redirected to X files, each containing Y GC cycles.

The dump agent tokens can be used in the filename. See "Dump agent tokens" on page 335 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 84. For information about when to use this policy, see “When to use the Balanced garbage collection policy” on page 89.

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>
- Xmne<size>
- Xminf<percentage>
- Xmint<percentage>
- Xmns<size>
- Xmnx<size>
- Xm<size>
- Xmn<size>
- Xmnx<size>
- Xms<size>
- Xm<size>
- Xnoclassgc
- Xnocompactexplicitgc
- Xnuma:none
- Xsoftmx<size>
- Xsoftrefthreshold<number>
- Xverbosegclog[:<file> [, <X>,<Y>]]

A detailed description of these command-line options can be found in “Garbage Collector command-line options” on page 577.

The behavior of the following options is different when specified with \texttt{-Xgcpolicy:balanced}:

\texttt{-Xcompactgc}

Compaction occurs when a System.gc() call is received (default). Compaction always occurs on all other collection types.

\texttt{-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 \texttt{-Xgcpolicy:balanced}:

- \texttt{-Xconcurrentbackground<number>}
- \texttt{-Xconcurrentlevel<number>}
- \texttt{-Xconcurrentslack<size>}
- \texttt{-Xcometer:<soa | loa | dynamic>}
- \texttt{-Xdisablestringconstantgc}
- \texttt{-Xenablestringconstantgc}
- \texttt{-Xloa}
- \texttt{-Xloainitial<percentage>}
- \texttt{-Xloamaximum<percentage>}
- \texttt{-Xloaminimum<percentage>}
- \texttt{-Xmo<size>}
- \texttt{-Xmoi<size>}
- \texttt{-Xmos<size>}
- \texttt{-Xmr<size>}
- \texttt{-Xmrx<size>}
- \texttt{-Xnoloa}
- \texttt{-Xnopartialcompactgc} (deprecated)
• `-Xpartialcompactgc` (deprecated)

A detailed description of these command-line options can be found in “Garbage Collector command-line options” on page 577.

---

## JVM messages

Messages are issued by the Java technology 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 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 virtual machine messages have the following format:

`JVMTYPENUM&`

where:

- `JVM` is a standard prefix.
- `TYPE` refers to the JVM subcomponent that issued the message.
- `NUM` is a unique numerical number.
- `&` is one of the following codes:
  - `I` - Information message
  - `W` - Warning message
  - `E` - Error message

These messages can help you with problem determination. Refer to diagnostic information for more detailed information about diagnosing problems with the IBM virtual machine.

By default, all error and some information messages are routed to the system log and also written to stderr or stdout. The specific information messages are `JVMDUMP039I`, `JVMDUMP032I`, and `JVMDUMP033I`, which provide valuable additional information about dumps produced by the JVM. To route additional message types to the system log, or turn off message logging to the system log, use the `-Xlog` option. The `-Xlog` option does not affect messages written to the standard error stream (stderr). See “JVM command-line options” on page 549.

---

## 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 (/usr/sbin/syslogd). Logged messages are written to the syslog file that is configured in /etc/syslog.conf. If the syslog daemon is not running, logged messages are lost.

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.

For more information about AIX logging, see: Error-logging overview.

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 VM Messages Guide: IBM VM 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 297 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.

org.omg.CORBA.BAD_OPERATION

A method was called on an object that does not support that method.
For example, this exception is thrown if the extract() method is called on an IBM ORB-generated Helper class, but the type of the provided Any object does not match the type that is expected by the Helper class.

For example, consider the following interface definition language (IDL) definition:

```idl
struct StructuredEvent {
    string header;
    any remainder_of_body;
};

struct ExpectedRemainderOfBody {
    string desc1;
    string desc2;
};

typedef string UnexpectedActualRemainderOfBody;
```

Assuming the server is designed to return a StructuredEvent object as a result of an operation that is called by the client, the type of the remainder_of_body field should be an ExpectedRemainderOfBody object. The client side then calls the ExpectedRemainderOfBodyHelper.extract() method by passing the remainder_of_body field.

If the server does not behave as designed but instead puts an UnexpectedActualRemainderOfBody object in the remainder_of_body field, this exception is thrown when the client calls the ExpectedRemainderOfBodyHelper.extract() method.

org.omg.CORBA.BAD_PARAM
An invalid parameter was passed. For example, the parameter is out of range, or is a null values or pointer.

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
<th>Explanation</th>
<th>User response</th>
</tr>
</thead>
<tbody>
<tr>
<td>4942f216</td>
<td>NULL_PI_NAME</td>
<td>The name() method of the interceptor input parameter returned a null string.</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4942f308</td>
<td>CONN_CLOSE_REBIND</td>
<td>A communication failure occurred during an operation. This exception is raised after the client sends a request, but before the server returns a reply.</td>
<td>Ensure that the completion status that is associated with the minor code is NO, then reissue the request.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
necessary. If the problem occurs again, enable ORB, network tracing, or both, to determine the cause of the failure.

**org.omg.CORBA.INTERNAL**

An internal ORB failure, such as corruption of internal data structures, occurred.

Explanation: An exception was caught in the NameService while the NamingContext.List() method was executing.

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.

**org.omg.CORBA.MARSHAL**

A request or reply is structurally invalid. For example, if method parameters are supplied that do not match the parameters defined for the method in the CORBA interface definition language (IDL), or if a message is a different size from the size listed within the message.

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.

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.

**org.omg.CORBA.NO_RESPONSE**

No response was received from the server. For example, this exception is thrown when a deferred asynchronous call is made, and the client attempts to get a result before the response is available.

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.

User response: None.

**org.omg.CORBA.OBJ_ADAPTER**

An object adapter detected a failure, such as a failed server registration due to the use of an existing registration name.

Explanation: A servant failed to connect to a server-side ORB.

User response: See the FFDC log for the cause of the problem, then try restarting the application.

**org.omg.CORBA.OBJECT_NOT_EXIST**

The referenced object does not exist. If the object existed at some point, it has now been deleted. Any remaining references to the object, for example references in code that receives the exception or proxy objects within an interoperability bridge, may also be deleted.
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:

**Explanation:**

4942fc01  
**LOCATE_UNKNOWN_OBJECT**

**Explanation:** The server has no knowledge of the object for which the client has asked in a locate request.

**User response:** Ensure that the remote object that is requested resides in the specified server and that the remote reference is up-to-date.

**org.omg.CORBA.TRANSIENT**

An object could not be reached. The object might or might not exist. For example, a connection cannot be established because the server is inactive.

**Explanation:**

4942fe02  
**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).

**User response:** As with all TRANSIENT exceptions, trying again or restarting the client or server might solve the problem. Ensure that the port and server host names are correct, and that the server is running and allowing connections. Also ensure that no firewall is blocking the connection, and that a route is available between client and server.

**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.

**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.

**Explanation:**

4942FE07  
**CONNECT_FAILURE_5**

**Explanation:** An attempt to connect to a server failed with both the direct and indirect IORs. Every client side handle to a server object (managed by the ClientDelegate reference) is set up with two IORs (object references) to reach the servant on the server. The first IOR is the direct IOR, which holds details of the server hosting the object. The second IOR is the indirect IOR, which holds a reference to a naming server that can be queried if the direct IOR fails.

**Note:** The root cause of 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).

**User response:** As with all TRANSIENT exceptions, trying again or restarting the client or server might solve the problem. Ensure that the port and server host names are correct, and that the server is running and allowing connections. Also ensure that no firewall is blocking the connection, and that a route is available between client and server.

**Explanation:**

4f4d0001  
**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 [https://www.omg.org/cgi-bin/doc?formal/99-10-07](https://www.omg.org/cgi-bin/doc?formal/99-10-07)

**User response:** Put the POA Manager into the active state if you want requests to be processed.

**Explanation:**

4f4dfe85  
**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.

**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.
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:

```shell
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:

```shell
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:

```shell
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), C:\mycode;C:\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 398. Note these restrictions.

**JAVA_FONTS=list of directories**
Set this environment variable to specify the font directory. Setting this variable is equivalent to setting the property 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 321.

**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 338.

`TMPDIR=<directory>`
This variable specifies an alternative temporary directory. This directory is used only when Javadumps and Heapdumps cannot be written to their target directories, or the current working directory.

This variable defaults to /tmp on Linux, z/OS, AIX, and i5/OS.

**Diagnostic options**

The following list describes the diagnostic options:

`IBM_COREDIR=<directory>`
Set this variable to specify an alternative location for system dumps, JIT 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.
The setting is controlled by a command-line parameter or an environment variable. The command-line parameter always takes precedence.

<table>
<thead>
<tr>
<th>JVM setting</th>
<th>Default</th>
<th>Setting affected by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Javadump</td>
<td>Enabled</td>
<td>ec</td>
</tr>
<tr>
<td>Heapdump</td>
<td>Disabled</td>
<td>ec</td>
</tr>
<tr>
<td>System dump</td>
<td>Enabled</td>
<td>ec</td>
</tr>
<tr>
<td>Snap traces</td>
<td>Enabled</td>
<td>ec</td>
</tr>
<tr>
<td>JIT dump</td>
<td>Enabled</td>
<td>ec</td>
</tr>
<tr>
<td>Verbose output</td>
<td>Disabled</td>
<td>c</td>
</tr>
<tr>
<td>Compressed references</td>
<td>Enabled for <code>-Xmx</code> values of less than or equal to 25 GB, otherwise disabled</td>
<td>ec</td>
</tr>
<tr>
<td>Boot classpath search</td>
<td>Disabled</td>
<td>c</td>
</tr>
<tr>
<td>JNI checks</td>
<td>Disabled</td>
<td>c</td>
</tr>
<tr>
<td>Remote debugging</td>
<td>Disabled</td>
<td>c</td>
</tr>
<tr>
<td>Strict conformance checks</td>
<td>Disabled</td>
<td>c</td>
</tr>
<tr>
<td>Quickstart</td>
<td>Disabled</td>
<td>c</td>
</tr>
<tr>
<td>Remote debug info server</td>
<td>Disabled</td>
<td>c</td>
</tr>
<tr>
<td>Reduced signaling</td>
<td>Disabled</td>
<td>c</td>
</tr>
<tr>
<td>Signal handler chaining</td>
<td>Enabled</td>
<td>c</td>
</tr>
<tr>
<td>Classpath</td>
<td>Not set</td>
<td>ec</td>
</tr>
<tr>
<td>Class data sharing</td>
<td>Disabled</td>
<td>c</td>
</tr>
<tr>
<td>Accessibility support</td>
<td>Enabled</td>
<td>e</td>
</tr>
<tr>
<td>JIT compiler</td>
<td>Enabled</td>
<td>ec</td>
</tr>
<tr>
<td>AOT compiler (AOT is not used by the JVM unless shared classes are also enabled)</td>
<td>Enabled</td>
<td>c</td>
</tr>
<tr>
<td>JIT debug options</td>
<td>Disabled</td>
<td>c</td>
</tr>
<tr>
<td>Java2D max size of fonts with algorithmic bold</td>
<td>14 point</td>
<td>e</td>
</tr>
<tr>
<td>Java2D use rendered bitmaps in scalable fonts</td>
<td>Enabled</td>
<td>e</td>
</tr>
<tr>
<td>Java2D freetype font rasterizing</td>
<td>Enabled</td>
<td>e</td>
</tr>
<tr>
<td>Java2D use AWT fonts</td>
<td>Disabled</td>
<td>e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JVM setting</th>
<th>AIX</th>
<th>IBM i</th>
<th>Linux</th>
<th>Windows</th>
<th>z/OS</th>
<th>Setting affected by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default locale</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>N/A</td>
<td>None</td>
<td>e</td>
</tr>
<tr>
<td>Time to wait before starting plug-in</td>
<td>N/A</td>
<td>N/A</td>
<td>Zero</td>
<td>N/A</td>
<td>N/A</td>
<td>e</td>
</tr>
<tr>
<td>Temporary directory</td>
<td>/tmp</td>
<td>/tmp</td>
<td>/tmp</td>
<td>c:\temp</td>
<td>/tmp</td>
<td>e</td>
</tr>
<tr>
<td>Plug-in redirection</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>N/A</td>
<td>None</td>
<td>e</td>
</tr>
<tr>
<td>IM switching</td>
<td>Disabled</td>
<td>Disabled</td>
<td>Disabled</td>
<td>N/A</td>
<td>Disabled</td>
<td>e</td>
</tr>
<tr>
<td>IM modifiers</td>
<td>Disabled</td>
<td>Disabled</td>
<td>Disabled</td>
<td>N/A</td>
<td>Disabled</td>
<td>e</td>
</tr>
<tr>
<td>JVM setting</td>
<td>AIX</td>
<td>IBM i</td>
<td>Linux</td>
<td>Windows</td>
<td>z/OS</td>
<td>Setting affected by</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>--------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>--------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Thread model</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Native</td>
<td>e</td>
</tr>
<tr>
<td>Initial stack size for Java Threads 32-bit. Use: -Xiss&lt;size&gt;</td>
<td>2 KB</td>
<td>2 KB</td>
<td>2 KB</td>
<td>2 KB</td>
<td>2 KB</td>
<td>c</td>
</tr>
<tr>
<td>Maximum stack size for Java Threads 32-bit. Use: -Xss&lt;size&gt;</td>
<td>320 KB (Previously 256 KB)</td>
<td>320 KB (Previously 256 KB)</td>
<td>320 KB (Previously 256 KB)</td>
<td>320 KB (Previously 256 KB)</td>
<td>320 KB (Previously 256 KB)</td>
<td>c</td>
</tr>
<tr>
<td>Stack size for OS Threads 32-bit. Use -Xms0&lt;size&gt;</td>
<td>256 KB</td>
<td>256 KB</td>
<td>256 KB</td>
<td>32 KB</td>
<td>256 KB</td>
<td>c</td>
</tr>
<tr>
<td>Initial stack size for Java Threads 64-bit. Use: -Xiss&lt;size&gt;</td>
<td>2 KB</td>
<td>N/A</td>
<td>2 KB</td>
<td>2 KB</td>
<td>2 KB</td>
<td>c</td>
</tr>
<tr>
<td>Maximum stack size for Java Threads 64-bit. Use: -Xss&lt;size&gt;</td>
<td>1024 KB (Previously 512 KB)</td>
<td>N/A</td>
<td>1024 KB (Previously 512 KB)</td>
<td>1024 KB (Previously 512 KB)</td>
<td>1024 KB (Previously 512 KB)</td>
<td>c</td>
</tr>
<tr>
<td>Stack size for OS Threads 64-bit. Use -Xms0&lt;size&gt;</td>
<td>256 KB</td>
<td>N/A</td>
<td>256 KB</td>
<td>256 KB</td>
<td>256 KB</td>
<td>c</td>
</tr>
<tr>
<td>Initial heap size. Use -Xms&lt;size&gt;</td>
<td>(Previously 4 MB)</td>
<td>(Previously 4 MB)</td>
<td>(Previously 4 MB)</td>
<td>(Previously 4 MB)</td>
<td>(Previously 4 MB)</td>
<td>c</td>
</tr>
<tr>
<td>Maximum Java heap size. Use -Xmx&lt;size&gt;</td>
<td>Half the available memory with a minimum of 16 MB and a maximum of 512 MB</td>
<td>2 GB</td>
<td>Half the available memory with a minimum of 16 MB and a maximum of 512 MB</td>
<td>Half the available memory with a minimum of 16 MB and a maximum of 512 MB See note.</td>
<td>Half the available memory with a minimum of 16 MB and a maximum of 512 MB</td>
<td>c</td>
</tr>
<tr>
<td>Page size for the Java object heap and code cache. For restrictions, see &quot;Configuring large page memory allocation&quot; on page 231</td>
<td>Operating system default</td>
<td>Operating system default</td>
<td>Architecture: • Linux on x86 and AMD64/EM64T: 2 MB • Linux on IBM Z: 1 MB • Other architectures: operating system default</td>
<td>Operating system default</td>
<td>1M pageable</td>
<td>c</td>
</tr>
</tbody>
</table>

“Available memory” is defined as being the smallest of two values:

- The real or “physical” memory.
- The `RLIMIT_AS` value.
Known issues and limitations

Known issues or limitations that you might encounter in specific system environments, or configurations.

The problems described in this topic might not be limitations with this release. 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:

**Web Start and Version 1.3 applications**

For the 64-bit Version 7 release on AIX architectures, Web Start does not support launching Version 1.3 applications.

**Note:** As announced in the [Oracle Java SE Support roadmap](https://www.oracle.com/technetwork/java/javase-support-roadmap-138269.html), support for Java plug-in and Web Start technology in Java SE 7 ends in July 2016. Accordingly, support is also removed from IBM SDK, Java Technology Edition, Version 7 Release 1

**Desktop API**

If one or more GNOME libraries are not available, the 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.

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 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 212.

2. Use the Java option -Xlp4K. For more information about the -Xlp option, see “JVM command-line options” on page 549.

The change has no affect if you are already using the -Xlp option to allocate the Java heap with large pages.

Changes to locale translation files

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

Large page request fails

There is no error message issued when the JVM is unable to honor the -Xlp request.

There are a number of reasons why the JVM cannot honor a large page request. For example, there might be insufficient large pages available on the system at the time of the request. To check whether the -Xlp request was honored, you can review the output from -verbose:gc. Look for the attributes requestedPageSize and pageSize in the -verbose:gc log file. The attribute requestedPageSize contains
the value specified by -Xlp. The attribute pageSize is the actual page size used by
the JVM.

ThreadMXBean thread CPU time might not be monotonic on SMP systems

On SMP systems, the times returned by ThreadMXBean.getThreadUserTime(),
ThreadMXBean.getThreadCpuTime(), ThreadMXBean.getCurrentThreadUserTime(),
and ThreadMXBean.getCurrentThreadCpuTime() might not increase monotonically
if the relevant thread migrates to a different processor.

UDP datagram socket failure

By default on AIX, the system-wide udp_sendspace setting is 9216 bytes. If you are
trying to send buffer data with a length greater than 9216 bytes, a UDP Datagram
socket failure occurs. You can increase the size of the buffer by using the
setSendBufferSize() function available in
DatagramSocket.socket.setSendBufferSize(SEND_SIZE);

Unexpected CertificateException

This release contains a security enhancement to correctly validate certificates on jar
files of applications. A CertificateException occurs for any applications in one of
the following scenarios:
• The application jar is not properly signed.
• The application jar has incorrect certificates.
• A certificate in the certificate chain is revoked.

To avoid these exceptions, make sure that your application jars are signed with
valid certificates before you upgrade from an earlier release. This issue relates to
APAR IV38456.

Unexpected application errors with RMI

If your application uses RMI and you experience unexpected errors after
upgrading to this release, the problem might be associated with a change to the
default value of the RMI property java.rmi.server.useCodebaseOnly. For more
information, see https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/
enhancements-7.html

Unexpected XSLT error on extension elements or extension
functions when security is enabled

Any attempt to use extension elements or extension functions when security is
enabled, results in a javax.xml.transform.TransformerException error during
XSLT processing.

The following XSLT message is generated when extension functions are used: Use
of the extension function '<method name>' is not allowed when Java security
is enabled. To override this, set the
com.ibm.xtq.processor.overrideSecureProcessing property to true. This
override only affects XSLT processing.

The following XSLT message is generated when extension elements are used: Use
of the extension element '<element name>' is not allowed when Java security
is enabled. To override this, set the
com.ibm.xtq.processor.overrideSecureProcessing property to true. This override only affects XSLT processing.

To allow extensions when security is enabled, set the com.ibm.xtq.processor.overrideSecureProcessing system property to true. For more information about this system property, see “-Dcom.ibm.xtq.processor.overrideSecureProcessing” on page 536.

**Default behaviour change when running unsigned or self-signed applets**

Unsigned or self-signed applets no longer run by default because the security settings are changed to high. If you want to run unsigned or self-signed applets, you must change the security settings to medium. You can make this change in the Java Control Panel. Select the **Security** tab and move the slider to adjust the security level.

**Support for virtualization software**

This release is tested with a number of virtualized server products.

This release has been tested with the following virtualization software:

**Table 23. Virtualization software tested**

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Architecture</th>
<th>Server virtualization</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>IBM Z</td>
<td>PR/SM™</td>
<td>z13, z10™, z11, z196, zEC12</td>
</tr>
<tr>
<td>IBM</td>
<td>IBM Z</td>
<td>z/VM®</td>
<td>6.1, 6.2</td>
</tr>
<tr>
<td>IBM</td>
<td>IBM Z</td>
<td>KVM for IBM z Systems</td>
<td>1.1.0</td>
</tr>
<tr>
<td>IBM</td>
<td>POWER</td>
<td>PowerVM® Hypervisor</td>
<td>Power 6, Power 7, Power 8</td>
</tr>
<tr>
<td>VMware</td>
<td>x86-64</td>
<td>VMware ESX and ESXi Server</td>
<td>5.5, 6.0</td>
</tr>
<tr>
<td>Microsoft</td>
<td>x86-64</td>
<td>Hyper-V</td>
<td>Server 2012</td>
</tr>
<tr>
<td>Docker, Inc</td>
<td>x86-64</td>
<td>Docker</td>
<td>V1.6 or later (see note)</td>
</tr>
</tbody>
</table>

**Note:** IBM supports all versions of the SDK that run in Docker containers, provided that the Docker images are based on supported operating systems. To find out which operating systems are supported for the IBM SDK, see https://www.ibm.com/support/knowledgecenter/SSYKE2_7.1.0/com.ibm.java.lnx.71.doc/user/supported_env_71.html.

**Default Swing key bindings**

The default key bindings that are used to change the "look and feel" of your application's GUI.

- **Standard components**
- **Structured components**
- **Menu, Toolbar, and ToolTip components**
- **Text components**
• Containers: Frames, Windows, Panes, and Icons

**Standard components**

**JButton**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate Forward</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Navigate backward</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
</tr>
<tr>
<td>Activate default</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
<tr>
<td>Activate any</td>
<td>Space bar</td>
<td>Space bar</td>
<td>Space bar</td>
</tr>
<tr>
<td>Activate any</td>
<td>Alt+Char accel. key, if defined</td>
<td>Alt+Char accel. key, if defined</td>
<td>Alt+Char accel. key, if defined</td>
</tr>
</tbody>
</table>

**JCheckBox**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate forward</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Navigate backward</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
</tr>
<tr>
<td>Navigate within group</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
</tr>
<tr>
<td>Check and Uncheck</td>
<td>Space bar</td>
<td>Space bar</td>
<td>Space bar</td>
</tr>
</tbody>
</table>

**JRadioButton**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate forward</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Navigate backward</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
</tr>
<tr>
<td>Navigate within group</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
</tr>
<tr>
<td>Check and Uncheck</td>
<td>Space bar</td>
<td>Space bar</td>
<td>Space bar</td>
</tr>
</tbody>
</table>

**JTetleButton**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate forward</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Navigate backward</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
</tr>
<tr>
<td>Navigate within group</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
</tr>
<tr>
<td>Check and Uncheck</td>
<td>Space bar</td>
<td>Space bar</td>
<td>Space bar</td>
</tr>
</tbody>
</table>

**Note:** Navigating with the Tab keys does not select the button you navigate to, but navigating with Arrow keys does select the button.

**JComboBox**

See also JList for other navigation keys.
### Action

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate out backward</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
</tr>
<tr>
<td>Post menu</td>
<td>Alt+Down arrow</td>
<td>Alt+Down arrow</td>
<td>Alt+Down arrow</td>
</tr>
<tr>
<td>Retract menu</td>
<td>Esc</td>
<td>Esc</td>
<td></td>
</tr>
<tr>
<td>Retract menu</td>
<td>Alt+Up arrow</td>
<td>Alt+Up arrow</td>
<td>Alt+Up arrow</td>
</tr>
<tr>
<td>Toggle menu up or down</td>
<td>Alt+Up arrow, Alt+Down arrow</td>
<td>Alt+Up arrow, Alt+Down arrow</td>
<td>Alt+Up arrow, Alt+Down arrow</td>
</tr>
<tr>
<td>Activate menu item</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
<tr>
<td>Jump to menu item without selecting it</td>
<td>Initial character of item</td>
<td>Initial character of item</td>
<td>Initial character of item</td>
</tr>
<tr>
<td>Move up or down</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
</tr>
</tbody>
</table>

### EditField

See JTextfield in [Text components](#).

### JList

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate out forward</td>
<td>Tab</td>
<td>Tab</td>
<td></td>
</tr>
<tr>
<td>Navigate out backward</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
</tr>
<tr>
<td>Move within list</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
</tr>
<tr>
<td>Move to beginning of list</td>
<td>Ctrl+Home</td>
<td>Ctrl+Home</td>
<td>Ctrl+Home</td>
</tr>
<tr>
<td>Move to end of list</td>
<td>Ctrl+End</td>
<td>Ctrl+End</td>
<td>Ctrl+End</td>
</tr>
<tr>
<td>Select all entries</td>
<td>Ctrl+A</td>
<td>Ctrl+A</td>
<td>Ctrl+A</td>
</tr>
<tr>
<td>Make a selection (this will clear the previous selection)</td>
<td>Space bar</td>
<td>Space bar</td>
<td>Space bar</td>
</tr>
<tr>
<td>Extend selection up</td>
<td>Shift+Up arrow</td>
<td>Shift+Up arrow</td>
<td>Shift+Up arrow</td>
</tr>
<tr>
<td>Extend selection down</td>
<td>Shift+Down arrow</td>
<td>Shift+Down arrow</td>
<td>Shift+Down arrow</td>
</tr>
<tr>
<td>Extend selection to top</td>
<td>Shift+Home</td>
<td>Shift+Home</td>
<td>Shift+Home</td>
</tr>
<tr>
<td>Extend selection to end</td>
<td>Shift+End</td>
<td>Shift+End</td>
<td>Shift+End</td>
</tr>
<tr>
<td>Block extend up</td>
<td>Shift+PgUp</td>
<td>Shift+PgUp</td>
<td>Shift+PgUp</td>
</tr>
<tr>
<td>Block extend down</td>
<td>Shift+PgDn</td>
<td>Shift+PgDn</td>
<td>Shift+PgDn</td>
</tr>
<tr>
<td>Block move</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
</tr>
</tbody>
</table>

### JSlider

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate out forward</td>
<td>Tab</td>
<td>Tab</td>
<td></td>
</tr>
<tr>
<td>Navigate out backward</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
</tr>
<tr>
<td>Increase value</td>
<td>Up arrow, Right arrow</td>
<td>Up arrow, Right arrow</td>
<td>Up arrow, Right arrow</td>
</tr>
<tr>
<td>Decrease value</td>
<td>Left arrow, Down arrow</td>
<td>Left arrow, Down arrow</td>
<td>Left arrow, Down arrow</td>
</tr>
<tr>
<td>Minimum value</td>
<td>Home</td>
<td>Home</td>
<td>Home</td>
</tr>
<tr>
<td>Maximum value</td>
<td>End</td>
<td>End</td>
<td>End</td>
</tr>
</tbody>
</table>
### Structured components

#### JTable

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Block increase</td>
<td>PgUp</td>
<td>PgUp</td>
<td>PgUp</td>
</tr>
<tr>
<td>Block decrease</td>
<td>PgDn</td>
<td>PgDn</td>
<td>PgDn</td>
</tr>
<tr>
<td>Navigate out forward</td>
<td>Ctrl+Tab</td>
<td>Ctrl+Tab</td>
<td>Ctrl+Tab</td>
</tr>
<tr>
<td>Navigate out backward</td>
<td>Ctrl+Shift+Tab</td>
<td>Ctrl+Shift+Tab</td>
<td>Ctrl+Shift+Tab</td>
</tr>
<tr>
<td>Move to next cell</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Move to previous cell</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
</tr>
<tr>
<td>Move to previous cell</td>
<td>Left arrow</td>
<td>Left arrow</td>
<td>Left arrow</td>
</tr>
<tr>
<td>Wrap to next row</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Wrap to previous row</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
</tr>
<tr>
<td>Block move vertical</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
</tr>
<tr>
<td>Block move left</td>
<td>Ctrl+PgUp</td>
<td>Ctrl+PgUp</td>
<td>Ctrl+PgUp</td>
</tr>
<tr>
<td>Block move right</td>
<td>Ctrl+PgDn</td>
<td>Ctrl+PgDn</td>
<td>Ctrl+PgDn</td>
</tr>
<tr>
<td>Block extend vertical</td>
<td>Shift+PgUp, Shift+PgDn</td>
<td>Shift+PgUp, Shift+PgDn</td>
<td>Shift+PgUp, Shift+PgDn</td>
</tr>
<tr>
<td>Block extend left</td>
<td>Ctrl+Shift+PgUp</td>
<td>Ctrl+Shift+PgUp</td>
<td>Ctrl+Shift+PgUp</td>
</tr>
<tr>
<td>Block extend right</td>
<td>Ctrl+Shift+PgDn</td>
<td>Ctrl+Shift+PgDn</td>
<td>Ctrl+Shift+PgDn</td>
</tr>
<tr>
<td>Move to first cell in row</td>
<td>Home</td>
<td>Home</td>
<td>Home</td>
</tr>
<tr>
<td>Move to last cell in row</td>
<td>End</td>
<td>End</td>
<td>End</td>
</tr>
<tr>
<td>Move to first cell in table</td>
<td>Ctrl+Home</td>
<td>Ctrl+Home</td>
<td>Ctrl+Home</td>
</tr>
<tr>
<td>Move to last cell in table</td>
<td>Ctrl+End</td>
<td>Ctrl+End</td>
<td>Ctrl+End</td>
</tr>
<tr>
<td>Select all cells</td>
<td>Ctrl+A</td>
<td>Ctrl+A</td>
<td>Ctrl+A</td>
</tr>
<tr>
<td>Deselect current selection</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
</tr>
<tr>
<td>Deselect current selection</td>
<td>Ctrl+Up arrow, Ctrl+Down arrow</td>
<td>Ctrl+Up arrow, Ctrl+Down arrow</td>
<td>PgUp, PgDn</td>
</tr>
<tr>
<td>Deselect current selection</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
</tr>
<tr>
<td>Deselect current selection</td>
<td>Ctrl+PgUp, Ctrl+PgDn</td>
<td>Ctrl+PgUp, Ctrl+PgDn</td>
<td>Ctrl+PgUp, Ctrl+PgDn</td>
</tr>
<tr>
<td>Deselect current selection</td>
<td>Home, End</td>
<td>Home, End</td>
<td>Home, End</td>
</tr>
<tr>
<td>Deselect current selection</td>
<td>Ctrl+Home, Ctrl+End</td>
<td>Ctrl+Home, Ctrl+End</td>
<td>Ctrl+Home, Ctrl+End</td>
</tr>
<tr>
<td>Extend selection one row</td>
<td>Shift+Up arrow, Shift+Down arrow</td>
<td>Shift+Up arrow, Shift+Down arrow</td>
<td>Shift+Up arrow, Shift+Down arrow</td>
</tr>
<tr>
<td>Extend selection one column</td>
<td>Shift+Left arrow, Shift+Right arrow</td>
<td>Shift+Left arrow, Shift+Right arrow</td>
<td>Shift+Left arrow, Shift+Right arrow</td>
</tr>
<tr>
<td>Extend selection to beginning or end of row</td>
<td>Shift+Home, Shift+End</td>
<td>Shift+Home, Shift+End</td>
<td>Shift+Home, Shift+End</td>
</tr>
<tr>
<td>Extend selection to beginning or end of column</td>
<td>Ctrl+Shift+Home, Ctrl+Shift+End</td>
<td>Ctrl+Shift+Home, Ctrl+Shift+End</td>
<td>Ctrl+Shift+Home, Ctrl+Shift+End</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>--------------------------</td>
<td>----------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Edit cell without overriding current contents</td>
<td>F2</td>
<td>F2</td>
<td>F2</td>
</tr>
<tr>
<td>Reset cell content prior to editing</td>
<td>Esc</td>
<td>Esc</td>
<td>Esc</td>
</tr>
</tbody>
</table>

**JTree**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate out forward</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Navigate out backward</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
</tr>
<tr>
<td>Expand entry</td>
<td>Right arrow</td>
<td>Right arrow</td>
<td>Right arrow</td>
</tr>
<tr>
<td>Collapse entry</td>
<td>Left arrow</td>
<td>Left arrow</td>
<td>Left arrow</td>
</tr>
<tr>
<td>Toggle expand and collapse for entry</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
<tr>
<td>Move up or down one entry</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
</tr>
<tr>
<td>Move to first entry</td>
<td>Home</td>
<td>Home</td>
<td>Home</td>
</tr>
<tr>
<td>Move to last visible entry</td>
<td>End</td>
<td>End</td>
<td>End</td>
</tr>
<tr>
<td>Block move vertical</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
</tr>
<tr>
<td>Block extend vertical</td>
<td>Shift+PgUp, Shift+PgDn</td>
<td>Shift+PgUp, Shift+PgDn</td>
<td>Shift+PgUp, Shift+PgDn</td>
</tr>
<tr>
<td>Select all</td>
<td>Ctrl+A</td>
<td>Ctrl+A</td>
<td>Ctrl+A</td>
</tr>
<tr>
<td>Select all</td>
<td>Ctrl+Forward Slash (/)</td>
<td>Ctrl+Forward Slash (/)</td>
<td>Ctrl+Forward Slash (/)</td>
</tr>
<tr>
<td>Deselect all</td>
<td>Ctrl+Backslash ()</td>
<td>Ctrl+Backslash ()</td>
<td>Ctrl+Backslash ()</td>
</tr>
<tr>
<td>Single select</td>
<td>Ctrl+Space bar</td>
<td>Ctrl+Space bar</td>
<td>Ctrl+Space bar</td>
</tr>
<tr>
<td>Extend selection up</td>
<td>Shift+Up arrow</td>
<td>Shift+Up arrow</td>
<td>Shift+Up arrow</td>
</tr>
<tr>
<td>Extend selection down</td>
<td>Shift+Down arrow</td>
<td>Shift+Down arrow</td>
<td>Shift+Down arrow</td>
</tr>
<tr>
<td>Extend selection to start of data</td>
<td>Shift+Home</td>
<td>Shift+Home</td>
<td>Shift+Home</td>
</tr>
<tr>
<td>Extend selection to end of data</td>
<td>Shift+End</td>
<td>Shift+End</td>
<td>Shift+End</td>
</tr>
</tbody>
</table>

**Menu, Toolbar, and ToolTip components**

**JMenuBar**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump to menubar</td>
<td>Alt</td>
<td>Alt</td>
<td>Alt</td>
</tr>
<tr>
<td>Navigate out</td>
<td>Esc</td>
<td>Esc</td>
<td>Esc</td>
</tr>
<tr>
<td>Navigate out</td>
<td>Alt</td>
<td>Alt</td>
<td>Alt</td>
</tr>
<tr>
<td>Navigate between items within menu</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
</tr>
<tr>
<td>Select first item (if no item selected)</td>
<td>F10</td>
<td>F10</td>
<td>F10</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------</td>
<td>----------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Select next item</td>
<td>Right arrow</td>
<td>Right arrow</td>
<td>Right arrow</td>
</tr>
<tr>
<td>Select previous item</td>
<td>Left arrow</td>
<td>Left arrow</td>
<td>Left arrow</td>
</tr>
<tr>
<td>Post menu</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
</tr>
<tr>
<td>Post menu</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
<tr>
<td>Post menu</td>
<td>Space bar</td>
<td>Space bar</td>
<td>Space bar</td>
</tr>
<tr>
<td>Un-post menu</td>
<td>Esc</td>
<td>Esc</td>
<td>Esc</td>
</tr>
<tr>
<td>Un-post menu</td>
<td>Alt</td>
<td>Alt</td>
<td>Alt</td>
</tr>
</tbody>
</table>

**JMenu**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Post menu</td>
<td>F10</td>
<td>F10</td>
<td>F10</td>
</tr>
<tr>
<td>Post submenu</td>
<td>Right arrow</td>
<td>Right arrow</td>
<td>Right arrow</td>
</tr>
<tr>
<td>Move to next item</td>
<td>Down arrow</td>
<td>Down arrow</td>
<td>Down arrow</td>
</tr>
<tr>
<td>Move to previous item</td>
<td>Up arrow</td>
<td>Up arrow</td>
<td>Up arrow</td>
</tr>
<tr>
<td>Retract menu</td>
<td>Esc</td>
<td>Esc</td>
<td>Esc</td>
</tr>
<tr>
<td>Retract submenu</td>
<td>Left arrow</td>
<td>Left arrow</td>
<td>Left arrow</td>
</tr>
<tr>
<td>Activate default</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
</tbody>
</table>

**JMenuItem**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate in</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
</tr>
<tr>
<td>Navigate out</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
</tr>
<tr>
<td>Activate item</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
<tr>
<td>Activate item</td>
<td>Space bar</td>
<td>Space bar</td>
<td>Space bar</td>
</tr>
<tr>
<td>Activate item</td>
<td>Alt+Char accel. key, if defined</td>
<td>Alt+Char accel. key, if defined</td>
<td>Alt+Char accel. key, if defined</td>
</tr>
<tr>
<td>Post submenu</td>
<td>Right arrow</td>
<td>Right arrow</td>
<td>Right arrow</td>
</tr>
<tr>
<td>Retract submenu</td>
<td>Left arrow</td>
<td>Left arrow</td>
<td>Left arrow</td>
</tr>
<tr>
<td>Retract submenu</td>
<td>Esc</td>
<td>Esc</td>
<td>Esc</td>
</tr>
</tbody>
</table>

**JCheckBoxMenuItem**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate in</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
</tr>
<tr>
<td>Navigate out</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
</tr>
<tr>
<td>Check or Uncheck item</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
</tbody>
</table>
### JRadioButtonMenuItem

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate in</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
</tr>
<tr>
<td>Navigate out</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
</tr>
<tr>
<td>Check or Uncheck item and retract menu</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
</tbody>
</table>

### JPopupMenu

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Post submenu</td>
<td>Right arrow</td>
<td>Right arrow</td>
<td>Right arrow</td>
</tr>
<tr>
<td>Close submenu</td>
<td>Left arrow</td>
<td>Left arrow</td>
<td>Left arrow</td>
</tr>
<tr>
<td>Retract menu</td>
<td>Esc</td>
<td>Esc</td>
<td>Esc</td>
</tr>
<tr>
<td>Move within menu</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
</tr>
<tr>
<td>Activate entry</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
<tr>
<td>Activate entry</td>
<td>Space bar</td>
<td>Space bar</td>
<td>Space bar</td>
</tr>
</tbody>
</table>

### JToolBar

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate out forward</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Navigate out backward</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
</tr>
<tr>
<td>Navigate within</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
</tr>
<tr>
<td>Activate item</td>
<td>Space bar</td>
<td>Space bar</td>
<td>Space bar</td>
</tr>
</tbody>
</table>

### JToolTip

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Post tip</td>
<td>Ctrl+F1</td>
<td>Ctrl+F1</td>
<td>Ctrl+F1</td>
</tr>
<tr>
<td>Retract tip</td>
<td>Esc</td>
<td>Esc</td>
<td>Esc</td>
</tr>
<tr>
<td>Retract tip</td>
<td>Ctrl+F1</td>
<td>Ctrl+F1</td>
<td>Ctrl+F1</td>
</tr>
</tbody>
</table>

## Text components

### JTextField

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate out forward</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Navigate out backward</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
</tr>
<tr>
<td>Move to previous or next character</td>
<td>Left arrow, Right arrow</td>
<td>Left arrow, Right arrow</td>
<td>Left arrow, Right arrow</td>
</tr>
<tr>
<td>Move to previous or next word</td>
<td>Ctrl+Left arrow, Ctrl+Right arrow</td>
<td>Ctrl+Left arrow, Ctrl+Right arrow</td>
<td>Ctrl+Left arrow, Ctrl+Right arrow</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>--------------------------</td>
<td>----------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Move to start or end of field</td>
<td>Home, End</td>
<td>Home, End</td>
<td>Home, End</td>
</tr>
<tr>
<td>Select all</td>
<td>Ctrl+A</td>
<td>Ctrl+A</td>
<td></td>
</tr>
<tr>
<td>Deselect all</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
<td>Arrow keys</td>
</tr>
<tr>
<td>Extend selection left or right</td>
<td>Shift+Left arrow, Shift+Right arrow</td>
<td>Shift+Left arrow, Shift+Right arrow</td>
<td>Shift+Left arrow, Shift+Right arrow</td>
</tr>
<tr>
<td>Extend selection to start or end</td>
<td>Shift+Home, Shift+End</td>
<td>Shift+Home, Shift+End</td>
<td>Shift+Home, Shift+End</td>
</tr>
<tr>
<td>Extend selection to previous or next word</td>
<td>Ctrl+Shift+Left arrow, Ctrl+Shift+Right arrow</td>
<td>Ctrl+Shift+Left arrow, Ctrl+Shift+Right arrow</td>
<td>Ctrl+Shift+Left arrow, Ctrl+Shift+Right arrow</td>
</tr>
<tr>
<td>Copy selection</td>
<td>Ctrl+C</td>
<td>Ctrl+C</td>
<td></td>
</tr>
<tr>
<td>Cut selection</td>
<td>Ctrl+X</td>
<td>Ctrl+X</td>
<td></td>
</tr>
<tr>
<td>Paste from clipboard</td>
<td>Ctrl+V</td>
<td>Ctrl+V</td>
<td></td>
</tr>
<tr>
<td>Delete next character</td>
<td>Delete</td>
<td>Delete</td>
<td>Delete</td>
</tr>
<tr>
<td>Delete previous character</td>
<td>Backspace</td>
<td>Backspace</td>
<td>Backspace</td>
</tr>
</tbody>
</table>

**Note:** The Ctrl+A, Ctrl+C, Ctrl+X and Ctrl+V combinations only work on JFormattedTextField objects.

**JPasswordField**

See JTextField in [text components](#) for information about how to navigate or move within the field and make selections.

**JTextArea and JEditorPane**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate in</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Navigate out forward</td>
<td>Ctrl+Tab</td>
<td>Ctrl+Tab</td>
<td>Ctrl+Tab</td>
</tr>
<tr>
<td>Navigate out backward</td>
<td>Ctrl+Shift+Tab</td>
<td>Ctrl+Shift+Tab</td>
<td>Ctrl+Shift+Tab</td>
</tr>
<tr>
<td>Move up or down one line</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
</tr>
<tr>
<td>Move left or right one character</td>
<td>Left arrow, Right arrow</td>
<td>Left arrow, Right arrow</td>
<td>Left arrow, Right arrow</td>
</tr>
<tr>
<td>Move to start or end of line</td>
<td>Home, End</td>
<td>Home, End</td>
<td>Home, End</td>
</tr>
<tr>
<td>Move to previous or next word</td>
<td>Ctrl+Left arrow, Ctrl+Right arrow</td>
<td>Ctrl+Left arrow, Ctrl+Right arrow</td>
<td>Ctrl+Left arrow, Ctrl+Right arrow</td>
</tr>
<tr>
<td>Move to start or end of text area</td>
<td>Ctrl+Home, Ctrl+End</td>
<td>Ctrl+Home, Ctrl+End</td>
<td>Ctrl+Home, Ctrl+End</td>
</tr>
<tr>
<td>Block move up or down</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
</tr>
<tr>
<td>Block move left</td>
<td>Ctrl+PgUp</td>
<td>Ctrl+PgUp</td>
<td>Ctrl+PgUp</td>
</tr>
<tr>
<td>Block move right</td>
<td>Ctrl+PgDn</td>
<td>Ctrl+PgDn</td>
<td>Ctrl+PgDn</td>
</tr>
<tr>
<td>Block extend up</td>
<td>Shift+PgUp</td>
<td>Shift+PgUp</td>
<td>Shift+PgUp</td>
</tr>
<tr>
<td>Block extend down</td>
<td>Shift+PgDn</td>
<td>Shift+PgDn</td>
<td>Shift+PgDn</td>
</tr>
<tr>
<td>Block extend left</td>
<td>Ctrl+Shift+PgUp</td>
<td>Ctrl+Shift+PgUp</td>
<td>Ctrl+Shift+PgUp</td>
</tr>
<tr>
<td>Block extend right</td>
<td>Ctrl+Shift+PgDn</td>
<td>Ctrl+Shift+PgDn</td>
<td>Ctrl+Shift+PgDn</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------</td>
<td>---------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Select all</td>
<td>Ctrl+A</td>
<td>Ctrl+A</td>
<td>Arrow keys</td>
</tr>
<tr>
<td>Deselect all</td>
<td>Arrow keys</td>
<td>Shift+Up arrow, Shift+Down arrow</td>
<td>Shift+Up arrow, Shift+Down arrow</td>
</tr>
<tr>
<td>Extend selection</td>
<td>Shift+Up arrow, Shift+Down arrow</td>
<td>Shift+Up arrow, Shift+Down arrow</td>
<td>Shift+Up arrow, Shift+Down arrow</td>
</tr>
<tr>
<td>Extend selection left or right</td>
<td>Shift+Left arrow, Shift+Right arrow</td>
<td>Shift+Left arrow, Shift+Right arrow</td>
<td>Shift+Left arrow, Shift+Right arrow</td>
</tr>
<tr>
<td>Extend selection to start or end of line</td>
<td>Shift+Home, Shift+End</td>
<td>Shift+Home, Shift+End</td>
<td>Shift+Home, Shift+End</td>
</tr>
<tr>
<td>Extend selection to start or end of text area</td>
<td>Ctrl+Shift+Home, Ctrl+Shift+End</td>
<td>Ctrl+Shift+Home, Ctrl+Shift+End</td>
<td>Ctrl+Shift+Home, Ctrl+Shift+End</td>
</tr>
<tr>
<td>Extend selection to previous or next word</td>
<td>Ctrl+Shift+Left arrow, Ctrl+Shift+Right arrow</td>
<td>Ctrl+Shift+Left arrow, Ctrl+Shift+Right arrow</td>
<td>Ctrl+Shift+Left arrow, Ctrl+Shift+Right arrow</td>
</tr>
<tr>
<td>Copy selection</td>
<td>Ctrl+C</td>
<td>Ctrl+C</td>
<td>Delete</td>
</tr>
<tr>
<td>Cut selection</td>
<td>Ctrl+X</td>
<td>Ctrl+X</td>
<td>Delete</td>
</tr>
<tr>
<td>Paste Selected Text</td>
<td>Ctrl+V</td>
<td>Ctrl+V</td>
<td>Backspace</td>
</tr>
<tr>
<td>Delete next character</td>
<td>Delete</td>
<td>Delete</td>
<td>Backspace</td>
</tr>
<tr>
<td>Delete previous character</td>
<td>Backspace</td>
<td>Backspace</td>
<td>Backspace</td>
</tr>
<tr>
<td>Insert line break</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
<tr>
<td>Insert tab</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
</tbody>
</table>

**JTextPane**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate in</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Navigate out</td>
<td>Ctrl+Tab</td>
<td>Ctrl+Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Navigate out backwards</td>
<td>Shift+Ctrl+Tab</td>
<td>Shift+Ctrl+Tab</td>
<td>Shift+Ctrl+Tab</td>
</tr>
<tr>
<td>Move up or down a line</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
</tr>
<tr>
<td>Move left or right a component or character</td>
<td>Left/Right arrow</td>
<td>Left/Right arrow</td>
<td>Left/Right arrow</td>
</tr>
<tr>
<td>Move up or down one vertical block</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
</tr>
<tr>
<td>Move to start or end of line</td>
<td>Home, End</td>
<td>Home, End</td>
<td>Home, End</td>
</tr>
<tr>
<td>Move to previous or next word</td>
<td>Ctrl+Left arrow, Ctrl+Right arrow</td>
<td>Ctrl+Left arrow, Ctrl+Right arrow</td>
<td>Ctrl+Left arrow, Ctrl+Right arrow</td>
</tr>
<tr>
<td>Move to start or end of data</td>
<td>Ctrl+Home, Ctrl+End</td>
<td>Ctrl+Home, Ctrl+End</td>
<td>Ctrl+Home, Ctrl+End</td>
</tr>
<tr>
<td>Select all</td>
<td>Ctrl+A</td>
<td>Ctrl+A</td>
<td>Shift+Up arrow</td>
</tr>
<tr>
<td>Extend selection up one line</td>
<td>Shift+Up arrow</td>
<td>Shift+Up arrow</td>
<td>Shift+Up arrow</td>
</tr>
<tr>
<td>Extend selection down one line</td>
<td>Shift+Down arrow</td>
<td>Shift+Down arrow</td>
<td>Shift+Down arrow</td>
</tr>
<tr>
<td>Extend selection to beginning of line</td>
<td>Shift+Home</td>
<td>Shift+Home</td>
<td>Shift+Home</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>--------------------------</td>
<td>----------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Extend selection to end of line</td>
<td>Shift+End</td>
<td>Shift+End</td>
<td>Shift+End</td>
</tr>
<tr>
<td>Extend selection to beginning of data</td>
<td>Ctrl+Shift+Home</td>
<td>Ctrl+Shift+Home</td>
<td>Ctrl+Shift+Home</td>
</tr>
<tr>
<td>Extend selection to end of data</td>
<td>Ctrl+Shift+End</td>
<td>Ctrl+Shift+End</td>
<td>Ctrl+Shift+End</td>
</tr>
<tr>
<td>Extend selection left</td>
<td>Shift+Left arrow</td>
<td>Shift+Left arrow</td>
<td>Shift+Left arrow</td>
</tr>
<tr>
<td>Extend selection right</td>
<td>Shift+Right arrow</td>
<td>Shift+Right arrow</td>
<td>Shift+Right arrow</td>
</tr>
<tr>
<td>Extend selection up one vertical block</td>
<td>Shift+PgUp</td>
<td>Shift+PgUp</td>
<td>Shift+PgUp</td>
</tr>
<tr>
<td>Extend selection down one vertical block</td>
<td>Shift+PgDn</td>
<td>Shift+PgDn</td>
<td>Shift+PgDn</td>
</tr>
<tr>
<td>Extend selection left one word</td>
<td>Ctrl+Shift+Left arrow</td>
<td>Ctrl+Shift+Left arrow</td>
<td>Ctrl+Shift+Left arrow</td>
</tr>
<tr>
<td>Extend selection right one word</td>
<td>Ctrl+Shift+Right arrow</td>
<td>Ctrl+Shift+Right arrow</td>
<td>Ctrl+Shift+Right arrow</td>
</tr>
<tr>
<td>Move to next HTML link or other focusable element</td>
<td>Ctrl+T (letter T, not Tab)</td>
<td>Ctrl+T (letter T, not Tab)</td>
<td>Ctrl+T (letter T, not Tab)</td>
</tr>
<tr>
<td>Move to previous HTML link or other focusable element</td>
<td>Ctrl+Shift+T (letter T, not Tab)</td>
<td>Ctrl+Shift+T (letter T, not Tab)</td>
<td>Ctrl+Shift+T (letter T, not Tab)</td>
</tr>
<tr>
<td>Activate an HTML link</td>
<td>Ctrl+Space bar</td>
<td>Ctrl+Space bar</td>
<td>Ctrl+Space bar</td>
</tr>
<tr>
<td>Navigate out backwards</td>
<td>Ctrl+Shift+Tab</td>
<td>Ctrl+Shift+Tab</td>
<td>Ctrl+Shift+Tab</td>
</tr>
<tr>
<td>Move up or down one line</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
</tr>
<tr>
<td>Move left or right one component or character</td>
<td>Left/Right arrow</td>
<td>Left/Right arrow</td>
<td>Left/Right arrow</td>
</tr>
<tr>
<td>Move up or down one vertical block</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
</tr>
<tr>
<td>Move to beginning or end of line</td>
<td>Home, End</td>
<td>Home, End</td>
<td>Home, End</td>
</tr>
<tr>
<td>Move to previous or next word</td>
<td>Ctrl+Left arrow, Ctrl+Right arrow</td>
<td>Ctrl+Left arrow, Ctrl+Right arrow</td>
<td>Ctrl+Left arrow, Ctrl+Right arrow</td>
</tr>
<tr>
<td>Move to beginning or end of data</td>
<td>Ctrl+Home, Ctrl+End</td>
<td>Ctrl+Home, Ctrl+End</td>
<td>Ctrl+Home, Ctrl+End</td>
</tr>
<tr>
<td>Move left or right one block</td>
<td>Ctrl+PgUp, Ctrl+PgDn</td>
<td>Ctrl+PgUp, Ctrl+PgDn</td>
<td>Ctrl+PgUp, Ctrl+PgDn</td>
</tr>
<tr>
<td>Select all</td>
<td>Ctrl+A</td>
<td>Ctrl+A</td>
<td>Ctrl+A</td>
</tr>
<tr>
<td>Extend selection up or down one line</td>
<td>Shift+Up arrow, Shift+Down arrow</td>
<td>Shift+Up arrow, Shift+Down arrow</td>
<td>Shift+Up arrow, Shift+Down arrow</td>
</tr>
<tr>
<td>Extend selection left or right one component or character</td>
<td>Shift+Left arrow, Shift+Right arrow</td>
<td>Shift+Left arrow, Shift+Right arrow</td>
<td>Shift+Left arrow, Shift+Right arrow</td>
</tr>
<tr>
<td>Extend selection to start or end of line</td>
<td>Shift+Home, Shift+End</td>
<td>Shift+Home, Shift+End</td>
<td>Shift+Home, Shift+End</td>
</tr>
<tr>
<td>Extend selection to start or end of data</td>
<td>Ctrl+Shift+Home, Ctrl+Shift+End</td>
<td>Ctrl+Shift+Home, Ctrl+Shift+End</td>
<td>Ctrl+Shift+Home, Ctrl+Shift+End</td>
</tr>
</tbody>
</table>
### Action

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Extend selection up or down one vertical block</td>
<td>Shift+PgUp, Shift+PgDn</td>
<td>Shift+PgUp, Shift+PgDn</td>
<td>Shift+PgUp, Shift+PgDn</td>
</tr>
<tr>
<td>Extend selection to previous or next word</td>
<td>Ctrl+Shift+Left arrow, Ctrl+Shift+Right arrow</td>
<td>Ctrl+Shift+Left arrow, Ctrl+Shift+Right arrow</td>
<td>Ctrl+Shift+Left arrow, Ctrl+Shift+Right arrow</td>
</tr>
<tr>
<td>Extend selection left or right one block</td>
<td>Ctrl+Shift+PgUp, Ctrl+Shift+PgDn</td>
<td>Ctrl+Shift+PgUp, Ctrl+Shift+PgDn</td>
<td>Ctrl+Shift+PgUp, Ctrl+Shift+PgDn</td>
</tr>
</tbody>
</table>

### Containers: Frames, Windows, Panes, and Icons

#### JDesktopPane

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate forward among open application windows, then desktop icons</td>
<td>Alt+Tab</td>
<td>Alt+Tab</td>
<td>Alt+Tab</td>
</tr>
<tr>
<td>Navigate forward among open application windows, then desktop icons</td>
<td>Alt+Esc</td>
<td>Alt+Esc</td>
<td>Alt+Esc</td>
</tr>
<tr>
<td>Navigate backward among open application windows, then desktop icons</td>
<td>Alt+Shift+Tab</td>
<td>Alt+Shift+Tab</td>
<td>Alt+Shift+Tab</td>
</tr>
<tr>
<td>Navigate forward among open associated windows</td>
<td>Ctrl+F6</td>
<td>Ctrl+F6</td>
<td>Ctrl+F6</td>
</tr>
<tr>
<td>Navigate backward among open associated windows</td>
<td>Ctrl+Shift+F6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Post menu of window options</td>
<td>Alt+Space bar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Post menu of window options</td>
<td>Shift+Esc</td>
<td></td>
<td>Shift+Esc</td>
</tr>
<tr>
<td>Post menu of window options</td>
<td>Ctrl+Space bar</td>
<td></td>
<td>Ctrl+Space bar</td>
</tr>
<tr>
<td>Open window (when iconized)</td>
<td>Enter</td>
<td></td>
<td>Enter</td>
</tr>
</tbody>
</table>

#### JOptionPane

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Retract dialog</td>
<td>Esc</td>
<td>Esc</td>
<td>Esc</td>
</tr>
<tr>
<td>Activate the default button (if defined)</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
</tbody>
</table>

#### JDialog

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Retract dialog</td>
<td>Esc</td>
<td>Esc</td>
<td>Esc</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>--------------------------</td>
<td>---------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Activate the default button (if defined)</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
</tbody>
</table>

### JScrollPane

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate out forward</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Navigate out backward</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
</tr>
<tr>
<td>Move up or down</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
</tr>
<tr>
<td>Move left or right</td>
<td>Left arrow, Right arrow</td>
<td>Left arrow, Right arrow</td>
<td>Left arrow, Right arrow</td>
</tr>
<tr>
<td>Move to start or end of data</td>
<td>Ctrl+Home, Ctrl+End</td>
<td>Ctrl+Home, Ctrl+End</td>
<td>Ctrl+Home, Ctrl+End</td>
</tr>
<tr>
<td>Block move up or down</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
<td>PgUp, PgDn</td>
</tr>
<tr>
<td>Block move right</td>
<td>Ctrl+PgDn</td>
<td>Ctrl+PgDn</td>
<td>Ctrl+PgDn</td>
</tr>
<tr>
<td>Block move left</td>
<td>Ctrl+PgUp</td>
<td>Ctrl+PgUp</td>
<td>Ctrl+PgUp</td>
</tr>
</tbody>
</table>

### JSplitPane

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate out forward</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Navigate out forward</td>
<td>Ctrl+Tab</td>
<td>Ctrl+Tab</td>
<td>Ctrl+Tab</td>
</tr>
<tr>
<td>Navigate out backward</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
<td>Shift+Tab</td>
</tr>
<tr>
<td>Navigate out backward</td>
<td>Ctrl+Shift+Tab</td>
<td>Ctrl+Shift+Tab</td>
<td>Ctrl+Shift+Tab</td>
</tr>
<tr>
<td>Move between panes</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Move between panes</td>
<td>F6</td>
<td>F6</td>
<td>F6</td>
</tr>
<tr>
<td>Move to splitter bar</td>
<td>F8</td>
<td>F8</td>
<td>F8</td>
</tr>
<tr>
<td>Resize pane vertical</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
</tr>
<tr>
<td>Resize pane horizontal</td>
<td>Left arrow, Right arrow</td>
<td>Left arrow, Right arrow</td>
<td>Left arrow, Right arrow</td>
</tr>
<tr>
<td>Resize to min or max</td>
<td>Home, End</td>
<td>Home, End</td>
<td>Home, End</td>
</tr>
</tbody>
</table>

### JTabbedPane

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate in</td>
<td>Tab</td>
<td>Tab</td>
<td>Tab</td>
</tr>
<tr>
<td>Navigate out</td>
<td>Ctrl+Tab</td>
<td>Ctrl+Tab</td>
<td>Ctrl+Tab</td>
</tr>
<tr>
<td>Move to tab left or right</td>
<td>Left arrow, Right arrow</td>
<td>Left arrow, Right arrow</td>
<td>Left arrow, Right arrow</td>
</tr>
<tr>
<td>Move to tab above or below</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
<td>Up arrow, Down arrow</td>
</tr>
<tr>
<td>Move from tab to page</td>
<td>Ctrl+Down arrow</td>
<td>Ctrl+Down arrow</td>
<td>Ctrl+Down arrow</td>
</tr>
<tr>
<td>Move from page to tab</td>
<td>Ctrl+Up arrow</td>
<td>Ctrl+Up arrow</td>
<td>Ctrl+Up arrow</td>
</tr>
<tr>
<td>Move from page to previous page</td>
<td>Ctrl+PgUp</td>
<td>Ctrl+PgUp</td>
<td>Ctrl+PgUp</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------</td>
<td>-----------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Move from page to next page</td>
<td>Ctrl+PgDn</td>
<td>Ctrl+PgDn</td>
<td>Ctrl+PgDn</td>
</tr>
</tbody>
</table>

### JFrame

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigate out</td>
<td>Alt+Esc</td>
<td>Alt+Esc</td>
<td>Alt+Esc</td>
</tr>
<tr>
<td>Display window menu</td>
<td>Alt+Space bar</td>
<td>Alt+Space bar</td>
<td>Alt+Space bar</td>
</tr>
<tr>
<td>Activate the default button (if defined)</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
</tbody>
</table>

### JInternalFrame

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Open (Restore)</td>
<td>Ctrl+F5</td>
<td>Ctrl+F5</td>
<td>Ctrl+F5</td>
</tr>
<tr>
<td>Open (Restore)</td>
<td>Alt+F5</td>
<td>Alt+F5</td>
<td>Alt+F5</td>
</tr>
<tr>
<td>Open (Restore)</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
<tr>
<td>Close</td>
<td>Ctrl+F5</td>
<td>Ctrl+F5</td>
<td>Ctrl+F5</td>
</tr>
<tr>
<td>Close</td>
<td>Alt+F5</td>
<td>Alt+F5</td>
<td>Alt+F5</td>
</tr>
<tr>
<td>Move</td>
<td>Ctrl+F7</td>
<td>Ctrl+F7</td>
<td>Ctrl+F7</td>
</tr>
<tr>
<td>Move</td>
<td>Alt+F7</td>
<td>Alt+F7</td>
<td>Alt+F7</td>
</tr>
<tr>
<td>Resize</td>
<td>Ctrl+F8</td>
<td>Ctrl+F8</td>
<td>Ctrl+F8</td>
</tr>
<tr>
<td>Resize</td>
<td>Alt+F8</td>
<td>Alt+F8</td>
<td>Alt+F8</td>
</tr>
<tr>
<td>Minimize</td>
<td>Ctrl+F9</td>
<td>Ctrl+F9</td>
<td>Ctrl+F9</td>
</tr>
<tr>
<td>Minimize</td>
<td>Alt+F9</td>
<td>Alt+F9</td>
<td>Alt+F9</td>
</tr>
<tr>
<td>Display window menu</td>
<td>Alt+Space bar</td>
<td>Alt+Space bar</td>
<td>Alt+Space bar</td>
</tr>
<tr>
<td>Activate the default button (if defined)</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
</tbody>
</table>

### JWindow

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Activate the default button (if defined)</td>
<td>Enter</td>
<td>Enter</td>
<td>Enter</td>
</tr>
</tbody>
</table>
Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites.
websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data discussed herein is presented as derived under specific operating conditions. Actual results may vary.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:
Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at [www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml).

Intel, Intel logo, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.
Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see: (i) IBM’s Privacy Policy at [https://www.ibm.com/privacy](https://www.ibm.com/privacy); (ii) IBM’s Online Privacy Statement at [https://www.ibm.com/privacy/details](https://www.ibm.com/privacy/details) (in particular, the section entitled “Cookies, Web Beacons and Other Technologies”); and (iii) the “IBM Software Products and Software-as-a-Service Privacy Statement” at [https://www.ibm.com/software/info/product-privacy](https://www.ibm.com/software/info/product-privacy).