

IBM Developer Kit and Runtime Environment, Java
Technology Edition, Version 5.0
Version 5 Release 0

Diagnostics Guide



IBM Developer Kit and Runtime Environment, Java
Technology Edition, Version 5.0
Version 5 Release 0

Diagnostics Guide



Note

Before using this information and the product it supports, read the information in “Notices” on page 463.

Twelfth Edition (November 2009)

This edition applies to all the platforms that are included in the IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 5.0 and to all subsequent releases and modifications until otherwise indicated in new editions. Technical changes made for all the editions of this book are indicated by vertical bars to the left of the changes.

© Copyright IBM Corporation 2003, 2010.

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

Contents

Figures	ix
--------------------------	-----------

Tables	xi
-------------------------	-----------

About the <i>Diagnostics Guide</i>	xiii
---	-------------

What does the "Java Virtual Machine (JVM)" mean?	xiii
Who should read this book	xiv
Using this book.	xiv
Other sources of information	xv
Reporting problems	xv
Conventions and terminology	xv
How to send your comments	xvi
Contributors	xvi
Summary of changes	xvi

Part 1. Understanding the IBM Software Developers Kit (SDK) for Java 1

Chapter 1. The building blocks of the IBM Virtual Machine for Java. 3

Java application stack	4
Components of the IBM Virtual Machine for Java	4
JVM Application Programming Interface (API)	5
Diagnostics component	5
Memory management	5
Class loader	6
Interpreter	6
Platform port layer	6

Chapter 2. Memory management 7

Overview of memory management	7
Object allocation	7
Reachable objects	8
Garbage collection	8
Heap sizing problems	8
Allocation	9
Heap lock allocation	9
Cache allocation	9
Large Object Area	10
Detailed description of garbage collection	11
Mark phase	11
Sweep phase	14
Compaction phase	15
Subpool (AIX, Linux PPC and zSeries, z/OS and i5/OS only)	16
Reference objects	16
Final reference processing	17
JNI weak reference	17
Heap expansion	17
Heap shrinkage	18
Generational Concurrent Garbage Collector.	19
Tenure age.	20

Tilt ratio	20
How to do heap sizing	21
Initial and maximum heap sizes	21
Using verbose:gc	21
Using fine tuning options.	22
Interaction of the Garbage Collector with applications	22
How to coexist with the Garbage Collector	23
Root set	23
Thread local heap	23
Bug reports	23
Finalizers	24
Manually starting the Garbage Collector.	25
Frequently asked questions about the Garbage Collector	26

Chapter 3. Class loading 29

The parent-delegation model	29
Namespaces and the runtime package	30
Custom class loaders	31

Chapter 4. Class data sharing 33

Chapter 5. The JIT compiler. 35

JIT compiler overview	35
How the JIT compiler optimizes code.	36
Phase 1 - inlining	36
Phase 2 - local optimizations	36
Phase 3 - control flow optimizations	37
Phase 4 - global optimizations	37
Phase 5 - native code generation	37
Frequently asked questions about the JIT compiler	37

Chapter 6. Java Remote Method Invocation 39

The RMI implementation	39
Thread pooling for RMI connection handlers	40
Understanding distributed garbage collection	40
Debugging applications involving RMI	41

Chapter 7. The ORB 43

CORBA.	43
RMI and RMI-IIOP	44
Java IDL or RMI-IIOP?	44
RMI-IIOP limitations	44
Further reading	45
Examples of client-server applications	45
Interfaces	45
Remote object implementation (or servant)	45
Stubs and ties generation	46
Server code	47
Summary of major differences between RMI (JRMP) and RMI-IIOP	50
Using the ORB	51

How the ORB works	54
The client side	54
The server side	59
Additional features of the ORB	61
Portable object adapter	61
Fragmentation	63
Portable interceptors	63
Interoperable Naming Service (INS)	66

Chapter 8. The Java Native Interface (JNI) 69

Overview of JNI	69
The JNI and the Garbage Collector	70
Overview of JNI object references	70
JNI transitions	73
Copying and pinning	75
Using the isCopy flag	75
Using the mode flag	76
A generic way to use the isCopy and mode flags	76
Handling exceptions	76
Synchronization	77
Debugging the JNI	78
JNI checklist	79

Part 2. Submitting problem reports 81

Part 3. Problem determination . . . 83

Chapter 9. First steps in problem determination 85

Chapter 10. AIX problem determination 87

Setting up and checking your AIX environment	87
Enabling full AIX core files	88
General debugging techniques	89
AIX debugging commands	90
DBX Plug-in	99
Diagnosing crashes	100
Documents to gather	100
Locating the point of failure	101
Debugging hangs	102
AIX deadlocks	102
AIX busy hangs	102
Poor performance on AIX	105
Understanding memory usage	105
32- and 64-bit JVMs	105
The 32-bit AIX Virtual Memory Model	105
The 64-bit AIX Virtual Memory Model	106
Changing the Memory Model (32-bit JVM)	107
The native and Java heaps	107
The AIX 32-bit JVM default memory models	108
Monitoring the native heap	108
Native heap usage	109
Specifying MALLOCTYPE	110
Monitoring the Java heap	110
Receiving OutOfMemoryError exceptions	110
Is the Java or native heap exhausted?	111
Java heap exhaustion	111
Native heap exhaustion	111

AIX fragmentation problems	112
Submitting a bug report	113
Debugging performance problems	113
Finding the bottleneck	113
CPU bottlenecks	114
Memory bottlenecks	118
I/O bottlenecks	118
JVM heap sizing	118
JIT compilation and performance	119
Application profiling	119
MustGather information for AIX	119

Chapter 11. Linux problem determination 121

Setting up and checking your Linux environment	121
General debugging techniques	123
Using system dump tools	123
Examining process information	124
ldd	126
Tracing tools	126
Debugging with gdb	127
Diagnosing crashes	129
Debugging hangs	130
Debugging memory leaks	131
Debugging performance problems	131
Finding the bottleneck	131
CPU usage	131
Memory usage	132
Network problems	132
JVM heap sizing	133
JIT compilation and performance	133
Application profiling	133
MustGather information for Linux	134
Known limitations on Linux	136

Chapter 12. Windows problem determination 139

Setting up and checking your Windows environment	139
Windows 32-bit large address aware support	140
General debugging techniques	141
System dump	141
Diagnosing crashes in Windows	142
Data to send to IBM	142
Debugging hangs	143
Getting a dump from a hung JVM	143
Analyzing deadlocks	143
Debugging memory leaks	143
The Windows memory model	143
Classifying leaks	144
Tracing leaks	144
Using Heapdump to debug memory leaks	145
OutOfMemoryError creating a thread	145
Debugging performance problems	145
Finding the bottleneck	145
Windows systems resource usage	146
JVM heap sizing	146
JIT compilation and performance	146
Application profiling	146
MustGather information for Windows	146

Chapter 13. z/OS problem determination 149

Setting up and checking your z/OS environment	149
Maintenance.	149
LE settings	149
Environment variables	149
Private storage usage.	149
Setting up dumps	150
General debugging techniques.	151
Using IPCS commands	152
Using dbx	152
Interpreting error message IDs	152
Diagnosing crashes	153
Documents to gather	153
Determining the failing function	154
Working with TDUMPs using IPCS	156
Debugging hangs	159
The process is deadlocked	160
The process is looping	160
The process is performing badly	161
Understanding Memory Usage	161
Allocations to LE HEAP.	161
z/OS virtual storage	161
OutOfMemoryError exceptions	162
Debugging performance problems	163
Finding the bottleneck	163
z/OS systems resource usage	164
JVM heap sizing	164
JIT compilation and performance.	164
Application profiling	164
MustGather information for z/OS	164

Chapter 14. IBM i problem determination 167

Determining which VM is in use	167
Setting up your IBM Technology for Java	
Environment	167
Required Software and Licensing.	168
Configuring JAVA_HOME	168
Enabling i5/OS PASE core files	169
Setting environment variables for i5/OS PASE or QShell.	170
Determining the home directory for a user	171
Setting default Java command-line options	172
General debugging techniques.	173
Diagnosing problems at the command line	173
IBM i debugging commands	174
Work with Active Jobs (WRKACTJOB)	174
Work with Job (WRKJOB)	174
Work with System Status (WRKSYSSTS)	174
Work with Disk Status (WRKDSKSTS)	174
Process Status (ps).	175
Debugger (dbx).	175
Debugging performance problems	175
Analyzing CPU bottlenecks.	176
Analyzing memory problems	178
Analyzing I/O problems	179
Diagnosing crashes	182
Checking the system environment	182
Finding out about the Java environment	182

Detailed crash diagnosis.	182
Diagnosing hangs	182
i5/OS deadlocks	183
i5/OS busy hangs	183
Understanding memory usage.	183
The 32-bit i5/OS PASE Virtual memory model	183
The process and garbage-collected heaps	184
Monitoring the garbage-collected heap	184
Process heap usage	184
OutOfMemoryError exceptions	185
Garbage-collected heap exhaustion	185
Submitting a bug report.	185
Using dbx	186
Using the DBX Plug-in for Java	186
Important dbx usage notes and warnings	187
Using dbx to investigate a Java system dump	188
Starting dbx on a system dump (core.{date}.{time}.{pid}.dmp)	188

Chapter 15. Sun Solaris problem determination 189

Chapter 16. Hewlett-Packard SDK problem determination 191

Chapter 17. ORB problem determination 193

Identifying an ORB problem	193
Debug properties	194
ORB exceptions	195
Completion status and minor codes	196
Java security permissions for the ORB	197
Interpreting the stack trace	198
Description string	199
Interpreting ORB traces	199
Message trace	199
Comm traces	200
Client or server.	201
Service contexts	202
Common problems	202
ORB application hangs	202
Running the client without the server running before the client is started	203
Client and server are running, but not naming service.	204
Running the client with MACHINE2 (client) unplugged from the network	204
IBM ORB service: collecting data	205
Preliminary tests	205

Chapter 18. NLS problem determination 207

Overview of fonts	207
Font utilities.	208
Common NLS problem and possible causes	209

| Chapter 19. Attach API problem determination 211

Part 4. Using diagnostic tools . . . 215

Chapter 20. Overview of the available diagnostics . . . 217

Categorizing the problem	217
Summary of diagnostic information	217
Summary of cross-platform tooling	219
Heapdump analysis tooling	219
Cross-platform dump viewer	219
JVM TI tools	219
JVM PI tools	220
JPDA tools	220
DTFJ	220
Trace formatting	220
JVM RI	221

Chapter 21. Using dump agents . . . 223

Using the -Xdump option	223
Merging -Xdump agents.	225
Dump agents	227
Console dumps.	228
System dumps	228
Stack dumps	229
LE CEEDUMPs.	229
Tool option	230
Javadumps	231
Heapdumps	231
Snap traces	232
Dump events	232
Advanced control of dump agents	233
exec option	233
file option	234
filter option	234
opts option	235
Priority option	236
range option.	236
request option	237
defaults option	237
Dump agent tokens	238
Default dump agents	238
Removing dump agents	239
Dump agent environment variables	239
Signal mappings	241
Windows, Linux, AIX, and i5/OS specifics	241
z/OS specifics	242

Chapter 22. Using Javdump. . . . 245

Enabling a Javdump.	245
Triggering a Javdump	245
Interpreting a Javdump	247
Javdump tags	247
TITLE, GPINFO, and ENVINFO sections	248
Storage Management (MEMINFO)	250
Locks, monitors, and deadlocks (LOCKS)	252
Threads and stack trace (THREADS)	253
Classloaders and Classes (CLASSES)	255
Environment variables and Javdump	256

Chapter 23. Using Heapdump 257

Information for users of previous releases of	
Heapdump	257
Getting Heapdumps	257
Enabling text formatted ("classic") Heapdumps	258
Available tools for processing Heapdumps	258
Using -Xverbose:gc to obtain heap information	258
Environment variables and Heapdump.	258
Text (classic) Heapdump file format	259

Chapter 24. Using system dumps and the dump viewer 263

Overview of system dumps	263
System dump defaults	264
Overview of the dump viewer.	264
Using the dump extractor, jextract	264
Using the dump viewer, jdmpview	266
Dump viewer commands	267
General commands	267
Working with locks	269
Showing details of a dump.	269
Analyzing the memory	269
Working with classes	270
Working with objects	271
Generating Heapdumps	271
Working with trace	272
Example session	273

Chapter 25. Tracing Java applications and the JVM 283

What can be traced?	283
Types of tracepoint	284
Default tracing	284
Where does the data go?	285
Writing trace data to memory buffers	286
Writing trace data to a file	286
External tracing	287
Tracing to stderr	287
Trace combinations	287
Controlling the trace	287
Specifying trace options	288
Detailed descriptions of trace options	288
Using the Java API	303
Using the trace formatter	304
Determining the tracepoint ID of a tracepoint	305
Application trace	306
Implementing application trace	306
Using application trace at run time	308
Using method trace	309
Running with method trace	309
Untraceable methods	311
Examples of use	313
Example of method trace output	314

Chapter 26. JIT problem determination 317

Diagnosing a JIT problem	317
Disabling the JIT compiler	317
Selectively disabling the JIT compiler	318
Locating the failing method	319
Identifying JIT compilation failures	320
Performance of short-running applications	321

JVM behavior during idle periods	321
--	-----

Chapter 27. The Diagnostics Collector 323

Introduction to the Diagnostics Collector	323
Using the Diagnostics Collector	323
Collecting diagnostics from Java runtime problems	323
Verifying your Java diagnostics configuration.	325
Configuring the Diagnostics Collector	325
Diagnostics Collector settings	325
Known limitations.	327

Chapter 28. Garbage Collector diagnostics 329

How do the garbage collectors work?	329
Common causes of perceived leaks	329
Listeners	329
Hash tables	330
Static class data	330
JNI references	330
Objects with finalizers	330
-verbose:gc logging	330
Global collections	331
Garbage collection triggered by System.gc()	332
Allocation failures.	333
Scavenger collections.	334
Concurrent garbage collection	335
Timing problems during garbage collection	339
-Xtgc tracing	340
-Xtgc:backtrace	340
-Xtgc:compaction	341
-Xtgc:concurrent	341
-Xtgc:dump	341
-Xtgc:excessiveGC	342
-Xtgc:freelist.	342
-Xtgc:parallel	343
-Xtgc:references.	343
-Xtgc:scavenger.	343
-Xtgc:terse	344
Finding which methods allocated large objects	344

Chapter 29. Class-loader diagnostics 347

Class-loader command-line options	347
Class-loader runtime diagnostics	347
Loading from native code	348

Chapter 30. Shared classes diagnostics 351

Deploying shared classes	351
Cache naming	351
Cache access	352
Cache housekeeping	352
Cache performance	353
Compatibility between service releases	354
Nonpersistent shared cache cleanup.	355
Dealing with runtime bytecode modification	356
Potential problems with runtime bytecode modification.	356
Modification contexts.	357
SharedClassHelper partitions	357
Using the safemode option	357

Further considerations for runtime bytecode modification.	358
Understanding dynamic updates	358
Using the Java Helper API	361
SharedClassHelper API	362
Understanding shared classes diagnostics output	363
Verbose output	363
VerboseIO output	363
VerboseHelper output	364
printStats utility	364
printStatsAll utility	365
Debugging problems with shared classes	366
Using shared classes trace	366
Why classes in the cache might not be found or stored	366
Dealing with initialization problems.	367
Dealing with verification problems	369
Dealing with cache problems	369
Class sharing with OSGi ClassLoading framework	370

Chapter 31. Using the Reliability, Availability, and Serviceability Interface 371

Preparing to use JVMRI	371
Writing an agent	371
Registering a trace listener	372
Changing trace options	373
Starting the agent	373
Building the agent.	373
Agent design	374
JVMRI functions	374
API calls provided by JVMRI	375
CreateThread	375
DumpDeregister	375
DumpRegister	375
DynamicVerbosegc	376
GenerateHeapdump	376
GenerateJavacore	376
GetComponentDataArea.	376
GetRasInfo	377
InitiateSystemDump	377
InjectOutOfMemory	377
InjectSigSegv	377
NotifySignal.	378
ReleaseRasInfo	378
RunDumpRoutine.	378
SetOutOfMemoryHook	379
TraceDeregister	379
TraceDeregister50	379
TraceRegister	379
TraceRegister50.	380
TraceResume	380
TraceResumeThis	380
TraceSet	381
TraceSnap	381
TraceSuspend	381
TraceSuspendThis	381
RasInfo structure	382
RasInfo request types.	382
Intercepting trace data	382

The -Xtrace:external=<option>	382
Calling external trace.	383
Formatting	383

Chapter 32. Using the HPROF Profiler 385

Explanation of the HPROF output file	386
--	-----

Chapter 33. Using the JVMTI. 391

Chapter 34. Using the Diagnostic Tool

Framework for Java 393

Using the DTFJ interface	393
DTFJ example application	397

Chapter 35. Using JConsole 401

Chapter 36. Using the IBM Monitoring and Diagnostic Tools for Java - Health Center 405

Part 5. Appendixes 407

Appendix A. CORBA minor codes . . . 409

Appendix B. Environment variables 413

Displaying the current environment.	413
---	-----

Setting an environment variable	413
Separating values in a list	414
JVM environment settings	414
z/OS environment variables	418

Appendix C. Messages 419

DUMP messages	420
J9VM messages.	424
SHRC messages	427

Appendix D. Command-line options 439

Specifying command-line options.	439
General command-line options	440
System property command-line options	442
JVM command-line options.	444
-XX command-line options	452
JIT command-line options	452
Garbage Collector command-line options	453

Appendix E. Default settings for the JVM 461

Notices 463

Trademarks	464
----------------------	-----

Index 467

Figures

- | | | | |
|---|-----|-------------------------------------|-----|
| 1. Screenshot of ReportEnv tool | 140 | 2. DTFJ interface diagram | 396 |
|---|-----|-------------------------------------|-----|

Tables

1. Java subcomponents 293

About the *Diagnostics Guide*

The *Diagnostics Guide* tells you about how the IBM® Virtual Machine for Java™ works, debugging techniques, and the diagnostic tools that are available to help you solve problems with JVMs. It also gives guidance on how to submit problems to IBM.

What does the "Java Virtual Machine (JVM)" mean?

The Java Virtual machine (JVM) is the application that executes a Java program and it is included in the Java package.

The installable Java package supplied by IBM comes in two versions on Linux® and Windows® platforms:

- The Java Runtime Environment (JRE)
- The Java Software Development Kit (SDK)

The AIX®, z/OS®, and IBM i platforms ship only the SDK.

Note: IBM i is the integrated operating environment formerly referred to as IBM i5/OS®. The documentation might refer to IBM i as i5/OS.

The JRE provides runtime support for Java applications. The SDK provides the Java compiler and other development tools. The SDK includes the JRE.

The JRE (and, therefore, the SDK) includes a JVM. This is the application that executes a Java program. A Java program requires a JVM to run on a particular platform, such as Linux, z/OS, or Windows.

The IBM SDK, Version 5.0 contains a different implementation of the JVM and the Just-In-Time compiler (JIT) from most earlier releases of the IBM SDK, apart from the version 1.4.2 implementation on z/OS 64-bit and on AMD64/EM64T platforms. You can identify this implementation in the output from the `java -version` command, which gives these strings for the different implementations:

Implementation	Output
7	IBM J9 VM (build 2.6, JRE 1.7.0 IBM...
6	IBM J9 VM (build 2.4, J2RE 1.6.0 IBM...
5.0	IBM J9 VM (build 2.3, J2RE 1.5.0 IBM...
1.4.2 'classic'	Classic VM (build 1.4.2, J2RE 1.4.2 IBM...
1.4.2 on z/OS 64-bit and AMD64/EM64T platforms	IBM J9SE VM (build 2.2, J2RE 1.4.2 IBM...
i5/OS 'classic' 1.3.1	java version "1.3.1" Java(TM) 2 Runtime Environment, Standard Edition (build ...) Classic VM (build 1.3, build JDK-1.3, native threads, jitc_de)
i5/OS 'classic' 1.4.2	java version "1.4.2" Java(TM) 2 Runtime Environment, Standard Edition (build ...) Classic VM (build 1.4, build JDK-1.4, native threads, jitc_de)
i5/OS 'classic' 1.5.0	java version "1.5.0" Java(TM) 2 Runtime Environment, Standard Edition (build ...) Classic VM (build 1.5, build JDK-1.5, native threads, jitc_de)

Implementation	Output
i5/OS 'classic' 1.6.0	java version "1.6.0" Java(TM) SE Runtime Environment (build ...) Classic VM (build 1.6, build JDK-1.6, native threads, jitc_de)

For *Diagnostics Guides* that describe earlier IBM SDKs, see <http://www.ibm.com/developerworks/java/jdk/diagnosis/>. For earlier versions of the i5/OS Classic JVM, see the iSeries® Information Center at <http://publib.boulder.ibm.com/infocenter/iserics/v5r4/>.

Who should read this book

This book is for anyone who is responsible for solving problems with Java.

Using this book

Before you can use this book, you must have a good understanding of Software Developer Kits and the Runtime Environment.

This book is to be used with the IBM SDK and Runtime Environment Version 5.0. Check the full version of your installed JVM. If you do not know how to do this, see Chapter 9, “First steps in problem determination,” on page 85. Some of the diagnostic tools described in this book do not apply to earlier versions.

You can use this book in three ways:

- As an overview of how the IBM Virtual Machine for Java operates, with emphasis on the interaction with Java. Part 1, “Understanding the IBM Software Developers Kit (SDK) for Java,” on page 1 of the book provides this information. You might find this information helpful when you are designing your application.
- As straightforward guide to determining a problem type, collecting the necessary diagnostic data, and sending it to IBM. Part 2, “Submitting problem reports,” on page 81 and Part 3, “Problem determination,” on page 83 of the book provide this information.
- As the reference guide to all the diagnostic tools that are available in the IBM Virtual Machine for Java. This information is given in Part 4, “Using diagnostic tools,” on page 215 of the book.

The parts overlap in some ways. For example, Part 3, “Problem determination,” on page 83 refers to chapters that are in Part 4, “Using diagnostic tools,” on page 215 when those chapters describe the diagnostics data that is required. You will be able to more easily understand some of the diagnostics that are in Part 4, “Using diagnostic tools,” on page 215 if you read the appropriate chapter in Part 1, “Understanding the IBM Software Developers Kit (SDK) for Java,” on page 1.

The appendixes provide supporting reference information that is gathered into convenient tables and lists.

Other sources of information

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

- For the IBM SDKs, see the downloads at:

<http://www.ibm.com/developerworks/java/jdk/index.html>

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

<http://www.ibm.com/developerworks/java/>

- For Java documentation produced by Sun Microsystems, see:

<http://java.sun.com/reference/docs/index.html>

Reporting problems

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

The problem determination section provides guidance on diagnosing and correcting problems, including known workarounds. See Part 3, “Problem determination,” on page 83. If you cannot resolve the issue on your own, this section also tells you what data IBM service needs you to collect. Collect the data and send a problem report and associated data to IBM service, as described in Part 2, “Submitting problem reports,” on page 81.

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 Sun is intended as a reference to Sun Microsystems, Inc.

How to send your comments

Your feedback is important in helping to provide accurate and useful information.

If you have any comments about this *Diagnostics Guide*, you can send them by e-mail to jymcookbook@uk.ibm.com. Include the name of the *Diagnostics Guide*, the platform you are using, the version of your JVM, and, if applicable, the specific location of the text you are commenting on (for example, the title of the page).

Do not use this method for sending in bug reports on the JVM. For these, use the usual methods, as described in Part 2, “Submitting problem reports,” on page 81.

Contributors

This *Diagnostics Guide* has been put together by members of the Java Technology Center IBM development and service departments in Hursley, Bangalore, Austin, Toronto, Ottawa, and Rochester.

Summary of changes

This topic introduces what's new for this Version 5.0 Diagnostics Guide. This Version 5.0 *Diagnostics Guide* is based on the *Diagnostics Guide for z/OS64 and AMD64 platforms* for the IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2, SC34-6359-02.

For Version 5, additional platforms and new diagnostics features have been added.

To help people migrating from Version 1.4.2 or earlier, technical changes made for the first edition are still indicated by revision bars to the left of the changes.

For the Eleventh edition

The significant changes in this edition are:

- Description of **-Dcom.ibm.tools.attach.enable=yes** in “System property command-line options” on page 442.
- Description of Chapter 27, “The Diagnostics Collector,” on page 323 and the **-Xdiagnosticscollector[:settings=<filename>]** option in “JVM command-line options” on page 444.

For the tenth edition

The significant changes in this edition are:

- Description of a new dump agent event, called allocation that can be triggered when a Java object is allocated with a specified size. For more information about this event, see “Dump events” on page 232.
- Description of **-Dcom.ibm.nio.DirectByteBuffer.AggressiveMemoryManagement=true** in “System property command-line options” on page 442.
- Logging of data now occurs both to the console and to stderr or stdout.

For the ninth edition

The significant changes in this edition are:

- Addition of the `JavaReference` class to the DTFJ overview diagram, “Using the DTFJ interface” on page 393.

For the eighth edition

The significant changes in this edition are:

- Description of `-Xmxcl` in “JVM command-line options” on page 444 and of the problem it can resolve in “OutOfMemoryError exception when using delegated class loaders” on page 30.
- Description of `-Dsun.timezone.ids.oldermapping` in “System property command-line options” on page 442.

For the seventh edition

The significant changes in this edition are:

- Clarification of the use of `jextract` and `jdmpview` on z/OS in “Using the dump viewer, `jdmpview`” on page 266
- Correction to the state flags in “Threads and stack trace (THREADS)” on page 253

For the sixth edition

The significant changes for this edition are:

- The addition of `-Xdump:events=systhrow` in “Dump events” on page 232.
- A clarification of the JVM behavior during idle periods in “JVM behavior during idle periods” on page 321.
- A clarification of the results of setting the JIT parameter `count=0` in “Selectively disabling the JIT compiler” on page 318.

For the fifth edition

The significant changes for this edition are:

- Error information about `va_list` for z/OS, in “Debugging the JNI” on page 78.
- A new section, “Text (classic) Heapdump file format” on page 259.

For the fourth edition

The fourth edition was based on the third edition, the *Diagnostics Guide* for Java 2 Technology Edition Version 5.0, SC34-6650-02. Technical changes made for this edition are indicated by revision bars to the left of the changes.

The significant changes for the fourth edition are:

- Reorganization of chapters in Part 4, “Using diagnostic tools,” on page 215.

For the third edition

The significant changes for the third edition are:

- A new chapter, Chapter 14, “IBM i problem determination,” on page 167.

For the second edition

The significant changes for the second edition are:

- Improved information about debugging performance problems

- A new section, “Generational Concurrent Garbage Collector” on page 19
- Reviews of the AIX, Linux, Windows, and z/OS chapters in Part 3, “Problem determination,” on page 83
- Inclusion of Chapter 15, “Sun Solaris problem determination,” on page 189
- Inclusion of Chapter 16, “Hewlett-Packard SDK problem determination,” on page 191
- A new section, “Deploying shared classes” on page 351
- A new chapter, Chapter 34, “Using the Diagnostic Tool Framework for Java,” on page 393

For the first edition

The most significant changes are:

- New chapters:
 - Chapter 4, “Class data sharing,” on page 33
 - Chapter 10, “AIX problem determination,” on page 87, based on the AIX(R) chapter in the *Diagnostics Guide* for the IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2, SC34-6358-02. Revision bars indicate changes from that chapter.
 - Chapter 30, “Shared classes diagnostics,” on page 351
 - Chapter 32, “Using the HPROF Profiler,” on page 385
 - Chapter 33, “Using the JVMTI,” on page 391
 - Appendix C, “Messages,” on page 419
- Much changed chapters:
 - Chapter 2, “Memory management,” on page 7
 - Part 4, “Using diagnostic tools,” on page 215
 - Chapter 22, “Using Javadump,” on page 245
 - Chapter 28, “Garbage Collector diagnostics,” on page 329

Part 1. Understanding the IBM Software Developers Kit (SDK) for Java

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

The content provides:

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

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

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

The sections in this part are:

- Chapter 1, "The building blocks of the IBM Virtual Machine for Java," on page 3
- Chapter 2, "Memory management," on page 7
- Chapter 3, "Class loading," on page 29
- Chapter 4, "Class data sharing," on page 33
- Chapter 5, "The JIT compiler," on page 35
- Chapter 6, "Java Remote Method Invocation," on page 39
- Chapter 7, "The ORB," on page 43
- Chapter 8, "The Java Native Interface (JNI)," on page 69

Chapter 1. The building blocks of the IBM Virtual Machine for Java

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

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

The JVM specification also defines several other runtime characteristics.

All JVMs:

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

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

All the JVMs that are currently used commercially come with a supplementary compiler that takes bytecodes and produces platform-dependent machine code. This compiler works with the JVM to select parts of the Java program that could benefit from the compilation of bytecode, and replaces the JVM's virtualized interpretation of these areas of bytecode with concrete code. This is called just-in-time (JIT) compilation. IBM's JIT compiler is described in Chapter 5, "The JIT compiler," on page 35.

The IBM Virtual Machine for Java contains a number of private and proprietary technologies that distinguish it from other implementations of the JVM. In this release, IBM has made a significant change to the JVM and JIT compiler that were provided in earlier releases, while retaining full Java compliance. When you read this *Diagnostics Guide*, bear in mind that the particular unspecified behavior of this release of the JVM might be different to the behavior that you experienced in previous releases. Java programmers should not rely on the unspecified behavior of a particular JRE for this reason.

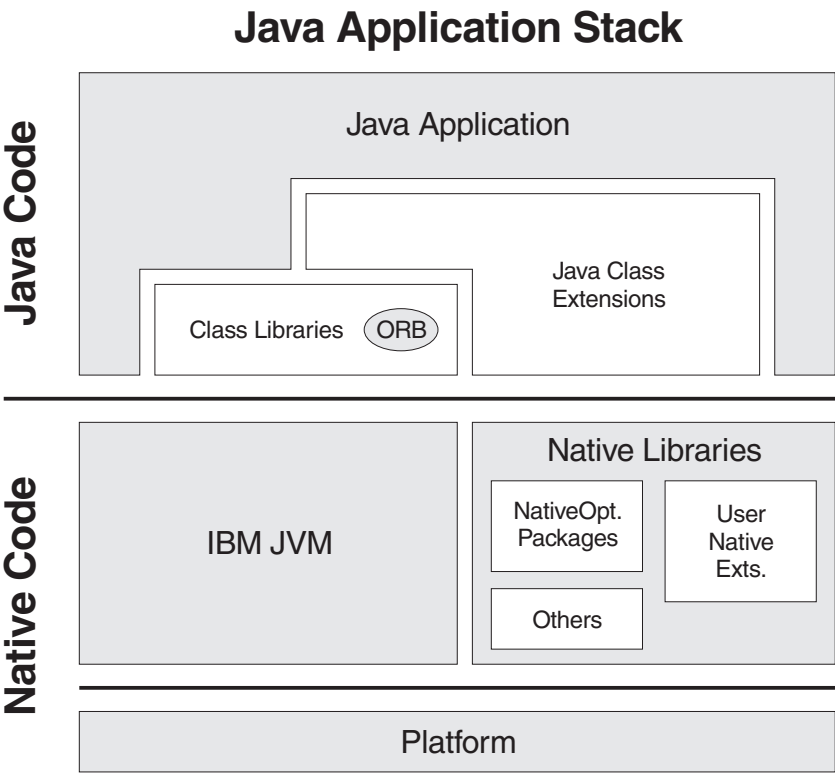
The diagnostic information in this guide discusses the characteristics of the IBM JRE that might affect the non-functional behavior of your Java program. This guide also provides information to assist you with tracking down problems and offers

advice, from the point of view of the JVM implementer, on how you can tune your applications. There are many other sources for good advice about Java performance, descriptions of the semantics of the Java runtime libraries, and tools to profile and analyze in detail the execution of applications.

Java application stack

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

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

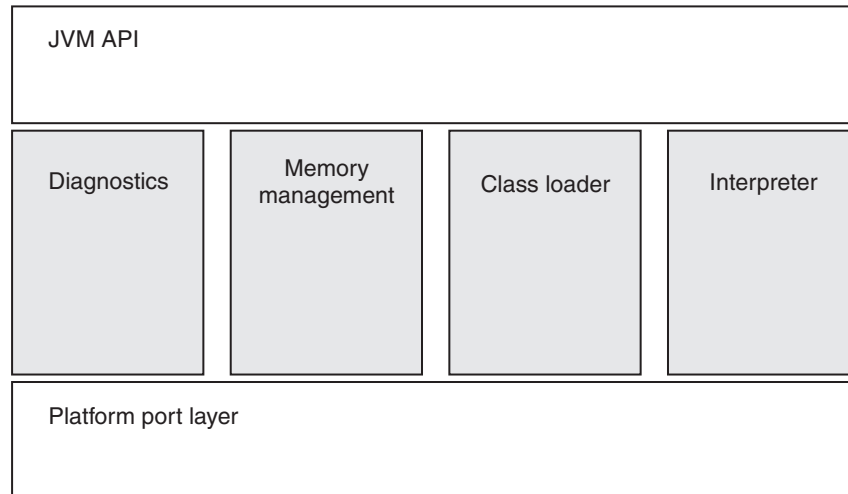


The JVM facilitates the invocation of native functions by Java applications and a number of well-defined Java Native Interface functions for manipulating Java from native code (for more information, see Chapter 8, “The Java Native Interface (JNI),” on page 69).

Components of the IBM Virtual Machine for Java

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

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



JVM Application Programming Interface (API)

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

Examples of this interaction include:

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

Diagnostics component

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

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

If a problem occurs in the field, it is possible to use the capabilities of the diagnostics component to trace the runtime function of the JVM and help to identify the cause of the problem. The diagnostics component can produce output selectively from various parts of the JVM and the JIT. Part 4, “Using diagnostic tools,” on page 215 describes various uses of the diagnostics component.

Memory management

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

Java programs run in a managed execution environment. When a Java program requires storage, the memory management component allocates the application a discrete region of unused memory. After the application no longer refers to the

storage, the memory management component must recognize that the storage is unused and reclaim the memory for subsequent reuse by the application or return it to the operating system.

The memory management component has several policy options that you can specify when you deploy the application. Chapter 2, “Memory management,” on page 7 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 Chapter 5, “The JIT compiler,” on page 35.

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.

Chapter 2. 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 9
- “Detailed description of garbage collection” on page 11
- “Generational Concurrent Garbage Collector” on page 19
- “How to do heap sizing” on page 21
- “Interaction of the Garbage Collector with applications” on page 22
- “How to coexist with the Garbage Collector” on page 23
- “Frequently asked questions about the Garbage Collector” on page 26

For detailed information about diagnosing Garbage Collector problems, see Chapter 28, “Garbage Collector diagnostics,” on page 329.

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

Overview of memory management

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

This section includes:

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

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

Object allocation

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

Every allocation requires a *heap lock* to be acquired to prevent concurrent thread access. To optimize this allocation, particular areas of the heap are dedicated to a thread, known as the TLH (thread local heap), and that thread can allocate from its TLH without having to lock out other threads. This technique delivers the best

possible allocation performance for small objects. Objects are allocated directly from a thread local heap. A new object is allocated from this cache without needing to grab the heap lock. All objects less than 512 bytes (768 bytes on 64-bit JVMs) are allocated from the cache. Larger objects are allocated from the cache if they can be contained in the existing cache. This cache is often referred to as the *thread local heap* or TLH.

Reachable objects

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

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

Garbage collection

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

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

Heap sizing problems

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

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

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

The frequency of garbage collections is too high until the heap reaches a steady state.

Use **`verbose:gc`** to determine the size of the heap at a steady state and set **`-Xms`** to this value.

The heap is fully expanded and the occupancy level is greater than 70%.

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

At 70% occupancy the frequency of garbage collections is too great.

Change the setting of **`-Xminf`**. The default is 0.3, which tries to maintain 30%

free space by expanding the heap. A setting of 0.4, for example, increases this free space target to 40%, and reduces the frequency of garbage collections.

Pause times are too long.

Try using `-Xgcpolicy:optavgpause`. This reduces the pause times and makes them more consistent when the heap occupancy rises. It does, however, reduce throughput by approximately 5%, although this value varies with different applications.

Here are some useful tips:

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

Allocation

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

Heap lock allocation

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

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

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

Cache allocation

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

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

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

The cache block is sometimes called a thread local heap (TLH). The size of the TLH varies from 2 KB 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 policies except **-Xgcpolicy:subpool** (for AIX, Linux PPC and zSeries®, z/OS, and i5/OS) but, if it is not used, it is shrunk to zero after a few collections. It can be disabled explicitly by specifying the **-Xnloa** command-line option.

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%)
- **-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.

Detailed description of 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.

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 above. 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.

By default, a platform with n processors also has $n-1$ new helper threads. The helper threads work with the master thread to complete the marking phase of garbage collection. You can override the default number of threads by using the **-Xgcthreads** option. If you specify a value of 1, the helper threads are not added. The **-Xgcthreads** option accepts any value greater than 0, but you gain nothing by setting it to more than $n-1$.

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 roots, i.e. stacks, JNI references, class static fields, and so on. 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 has to 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, to force a re-scan of part of the heap. The heap is divided into 512-byte sections and each section is allocated a one-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 0x01. A byte is used instead of a bit to eliminate contention; it allows marking of the cards using non-atomic operations. A stop-the-world (STW) collection is started when one of the following occurs:

- An allocation failure
- A System.gc
- Concurrent mark completes all the marking that it can do

The GC tries to start the concurrent mark phase so that it completes at the same time as the heap is exhausted. The GC does this by constant tuning of the parameters that govern the concurrent mark time. In the STW phase, the GC re-scans all roots and uses the marked cards to see what else must be retraced, and 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 referred to 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 CPU 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: <optthruput | optavgpause | gencon | subpool>

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

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. *Optthruput* is the default setting.

optavgpause

Enables concurrent mark with its default values. If you are having

problems with erratic application response times that are caused by normal garbage collections, you can reduce those problems at the cost of some throughput, by using the *optavgpause* option.

gencon

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

subpool

Disables concurrent mark. It uses an improved object allocation algorithm to achieve better performance when allocating objects on the heap. This option might improve performance on SMP systems with 16 or more processors. The *subpool* option is available only on AIX, Linux PPC and zSeries, z/OS, and i5/OS.

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.

Concurrent sweep

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

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

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

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

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

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

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

Compaction phase

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

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

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

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

- **-Xcompactgc** has been specified.
- Following the sweep phase, not enough free space is available to satisfy the allocation request.
- A `System.gc()` has been requested and the last allocation failure garbage collection did not compact or **-Xcompactexplicitgc** has been specified.

- At least half the previously available memory has been consumed by TLH allocations (ensuring an accurate sample) and the average TLH size falls below 1024 bytes
- Less than 5% of the active heap is free.
- Less than 128 KB of the heap is free.

Subpool (AIX, Linux PPC and zSeries, z/OS and i5/OS only)

Subpool is an improved GC policy for object allocation that is specifically targeted at improving the performance of object allocation on SMP systems with 16 or more processors. You start it with the **-Xgcpolicy:subpool** command-line option.

The subpool algorithm uses multiple free lists rather than the single free list used by **optavgpause** and **optthruput**. It tries to predict the size of future allocation requests based on earlier allocation requests. It recreates free lists at the end of each GC based on these predictions. While allocating objects on the heap, free chunks are chosen using a "best fit" method, as against the "first fit" method used in other algorithms. It also tries to minimize the amount of time for which a lock is held on the Java heap, thus reducing contention among allocator threads. Concurrent mark is disabled when subpool policy is used. Also, subpool policy uses a new algorithm for managing the Large Object Area (LOA). Hence, the **subpool** option might provide additional throughput optimization for some applications.

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 has not been marked for the previous 32 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 the previous 32 garbage collection cycles. You adjust the frequency of collection with the **-Xsoftrefthreshold** option. If there is a shortage of available storage, all soft references are cleared. All soft references are guaranteed to have been cleared before the `OutOfMemoryError` is thrown.

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

Final reference processing

The processing of objects that require finalization is more straightforward.

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

JNI weak reference

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

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

Heap expansion

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

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

- The Garbage Collector (GC) did not free enough storage to satisfy the allocation request.
- Free space is less than the minimum free space, which you can set by using the **-Xminf** parameter. The default is 30%.

- More than the maximum time threshold is being spent in garbage collection, set using the **-Xmaxt** parameter. The default is 13%.

The amount to expand the heap is calculated as follows:

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

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

You can set the minimum expansion amount using the **-Xmine** parameter. The default value is 1 MB. If the calculated required heap expansion is less than the value of **-Xmine**, the required heap expansion is increased to the value of **-Xmine**.

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

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

Heap shrinkage

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

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

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

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

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

On initialization, the JVM allocates the whole heap in a single contiguous area of virtual storage. The amount that is allocated is determined by the setting of the

-Xmx parameter. No virtual space from the heap is ever freed back to the native operating system. When the heap shrinks, it shrinks inside the original virtual space.

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

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

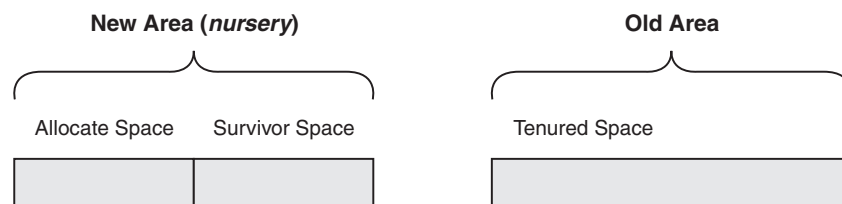
Where paging is supported, the GC allocates physical memory to the initial heap to the amount that is specified by the **-Xms** parameter. Additional memory is committed as the heap grows.

Generational Concurrent Garbage Collector

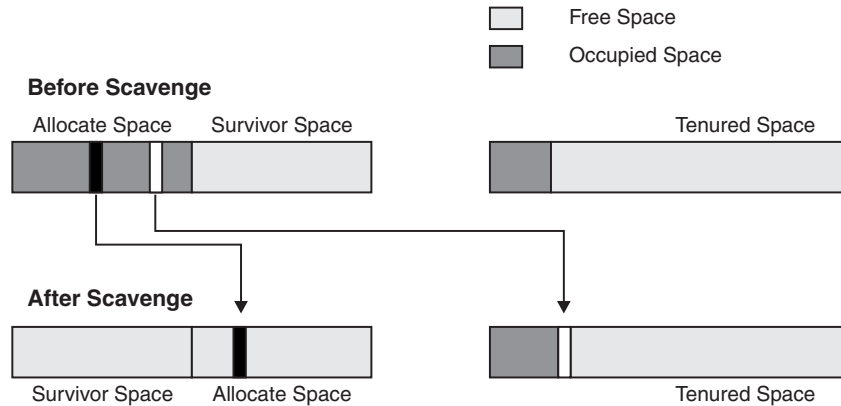
The Generational Concurrent Garbage Collector has been introduced in Java 5.0 from IBM. A generational garbage collection strategy is well suited to an application that creates many short-lived objects, as is typical of many transactional applications.

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

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



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



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

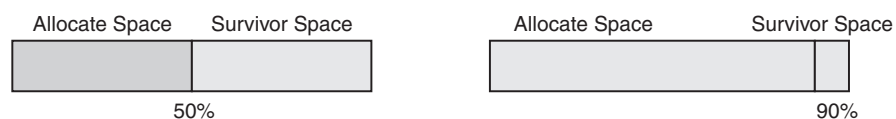
Tenure age

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

Tilt ratio

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

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



How to do heap sizing

You can do heap sizing to suit your requirements.

Generally:

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

Initial and maximum heap sizes

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

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

If the GC cannot find enough garbage, it runs compaction. If the GC finds enough garbage, or any of the other conditions for heap expansion are met (see “Heap expansion” on page 17), the GC expands the heap.

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

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

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

Using **verbose:gc**

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

The **-verbose:gc** output is fully described in Chapter 28, “Garbage Collector diagnostics,” on page 329. Switch on **-verbose:gc** and run up the application with no load. Check the heap size at this stage. This provides a rough guide to the start

size of the heap (**-Xms** option) that is needed. If this value is much larger than the defaults (see Appendix E, “Default settings for the JVM,” on page 461), 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 21.

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 17 and “Heap shrinkage” on page 18.

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 22, 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 22). This difference has significant consequences for finalizers (list item 7), which are described more fully in “Finalizers” on page 24.

Thread local heap

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

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

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

Bug reports

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

Here is an example of a regular bug report to Java service of the "Hello World" variety. A simple program allocates an object or objects, clears references to these objects, and then initiates a garbage collection cycle. The objects are not seen as collected. Typically, the objects are not collected because the application has attached a finalizer that does not run immediately.

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

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

See list item 1 in “Interaction of the Garbage Collector with applications” on page 22. 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 22.

Nature of finalizers

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

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

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

Finalizers and garbage collection

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

The way finalizers work, described in list item 7 in “Interaction of the Garbage Collector with applications” on page 22, 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 22. 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 22. The GC can honor a manual call; for example, through the `System.gc()` call. This call nearly always starts a garbage collection cycle, which is a heavy use of computer resources.

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

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

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

Frequently asked questions about the Garbage Collector

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

What are the default heap and native stack sizes?

See Appendix E, “Default settings for the JVM,” on page 461.

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

optthruput disables concurrent mark. If you do not have pause time problems (indicated by erratic application response times), you can expect to get the best throughput with this option.

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

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

subpool disables concurrent mark but employs an object allocation algorithm that is more suitable for multiple processor systems, commonly 16 processors or more. Applications that must scale on large systems can benefit from this policy. This policy is available on AIX, Linux PPC and zSeries, z/OS, and i5/OS only.

What is the default GC mode (`optavgpause`, `optthruput`, `gencon`, or `subpool`)?

optthruput - that is, generational collector and concurrent marking are off.

How many GC helper threads are created or “spawned”? What is their work?

A platform with *n* processors has *n*-1 helper threads. These threads work along with the main GC thread during:

- Parallel mark phase
- Parallel bitwise sweep phase
- Parallel compaction phase

You can control the number of GC helper threads with the **-Xgcthreads** option. Passing the **-Xgcthreads1** option to Java results in no helper threads at all.

You gain little by setting **-Xgcthreads** to more than *n*-1 other than possibly alleviating mark-stack overflows, if you suffer from them.

What is Mark Stack Overflow (MSO)? Why is MSO bad for performance?

Work packets are used for tracing all object reference chains from the roots. Each such reference that is found is pushed onto the mark stack so that it can be traced later. The number of work packets allocated is based on the heap size and therefore is finite and can overflow. This situation is called Mark Stack

Overflow (MSO). The algorithms to handle this situation are expensive in processing terms, and therefore MSO has a large impact on GC performance.

How can I prevent Mark Stack Overflow?

The following suggestions are not guaranteed to avoid MSO:

- Increase the number of GC helper threads using **-Xgcthreads** command-line option
- Decrease the size of the Java heap using the **-Xmx** setting.
- Use a small initial value for the heap or use the default.
- Reduce the number of objects the application allocates.
- If MSO occurs, you see entries in the verbose gc as follows:

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

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

When and why does the Java heap expand?

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

When does the Java heap shrink?

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

Does GC guarantee that it clears all the unreachable objects?

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

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

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

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

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

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

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

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

Chapter 3. Class loading

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

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

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

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

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

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

The parent-delegation model

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

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

```
Bootstrap
|
Extensions
|
Application
```

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

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

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

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

OutOfMemoryError exception when using delegated class loaders

When you use delegated class loaders, the JVM can create a large number of ClassLoader objects. For Java 5.0, up to and including Service Refresh 4, the number of class loaders that are permitted is limited and an OutOfMemoryError exception is thrown when this limit is exceeded. Use the `-Xmxcl` parameter to increase the number of class loaders allowed, to avoid this problem. On affected JVMs, the default limit is 8192, so if this problem occurs, increase this number, for example to 15000, by setting `-Xmxcl15000`, until the problem is resolved.

Examine the Javadump to recognize this problem. At the top of the Javadump there is an OutOfMemoryError exception like:

```
1TISIGINFO      Dump Event "systhrow" (00040000) Detail "java/lang/OutOfMemoryError" received
```

Further down, the current thread stack shows that a class loader is being loaded:

```
1XMCURTHDINFO   Current Thread Details
NULL           -----
3XMTHREADINFO   "ORB.thread.pool : 1" (TID:0x00000002ADD1FE300, sys_thread_t:0x00000002ACFE19998,
state:R, native ID:0x0000000042080960) prio=5
4XESTACKTRACE   at com/ibm/oti/vm/VM.initializeClassLoader(Native Method)
4XESTACKTRACE   at java/lang/ClassLoader.<init>(ClassLoader.java:120)
4XESTACKTRACE   at sun/reflect/DelegatingClassLoader.<init>(ClassDefiner.java:71)
```

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 above, 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 classloader is.

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

Chapter 4. Class data sharing

Class sharing in the IBM Version 5.0 SDK offers a transparent and dynamic means of sharing all loaded classes, both application classes and system classes, and placing no restrictions on Java Virtual Machines (JVMs) that are sharing the class data (unless runtime bytecode modification is being used).

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

You turn on shared classes with the **-Xshareclasses** command-line option. For reference information about **-Xshareclasses**, see "JVM command-line options" on page 444.

For diagnostics information about shared classes, see Chapter 30, "Shared classes diagnostics," on page 351.

Sharing all immutable class data for an application between multiple JVMs 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 the classes are already in memory and are already partially verified. Therefore, class sharing also benefits applications that regularly start new JVM instances doing similar tasks. The cost to populate an empty cache with a single JVM is minimal and, when more than one JVM is populating the cache concurrently, this activity is typically faster than both JVMs loading the classes from disk.

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 v5.0, apart from the Sun 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.
- 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 destroyed or until 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 bytecode that is modified at runtime 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.
- Only class data that does not change can be shared. Resources, objects, JIT compiled code, and similar items cannot be stored in the cache.

Chapter 5. The JIT compiler

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

JIT compiler overview

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

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

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

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

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

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

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

How the JIT compiler optimizes code

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

To help the JIT compiler analyze the method, its bytecodes are first reformulated in an internal representation called *trees*, which resembles machine code more closely than bytecodes. Analysis and optimizations are then performed on the trees of the method. At the end, the trees are translated into native code. The remainder of this section provides a brief overview of the phases of JIT compilation. For more information, see Chapter 26, “JIT problem determination,” on page 317.

The compilation consists of the following phases:

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

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

Phase 1 - inlining

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

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

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

Phase 2 - local optimizations

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

The optimizations include:

- Local data flow analyses and optimizations
- Register usage optimization
- Simplifications of Java idioms

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

Phase 3 - control flow optimizations

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

The optimizations are:

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

Phase 4 - global optimizations

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

The optimizations are:

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

Phase 5 - native code generation

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

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

Frequently asked questions about the JIT compiler

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

Can I disable the JIT compiler?

Yes. The JIT compiler is turned on by default, but you can set the appropriate command-line parameter to disable it. (See "Disabling the JIT compiler" on page 317.)

Can I use another vendor's JIT compiler?

No.

Can I use any version of the JIT compiler with the JVM?

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

Can the JIT compiler “decompile” methods?

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

Can I control the JIT compilation?

Yes. See Chapter 26, “JIT problem determination,” on page 317. In addition, advanced diagnostics are available to IBM engineers.

Can I dynamically control the JIT compiler?

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

How much memory does the code cache consume?

The JIT compiler uses memory intelligently. When the code cache is initialized, it consumes relatively little memory. As more methods are compiled into native code, the code cache is grown dynamically to accommodate the needs of the program. Space previously occupied by discarded or recompiled methods is reclaimed and reused. When the size of the code cache reaches a predefined upper limit, it stops growing. The JIT compiler will stop all future attempts to compile methods, to avoid exhausting the system memory and affecting the stability of the application or the operating system.

Chapter 6. Java Remote Method Invocation

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

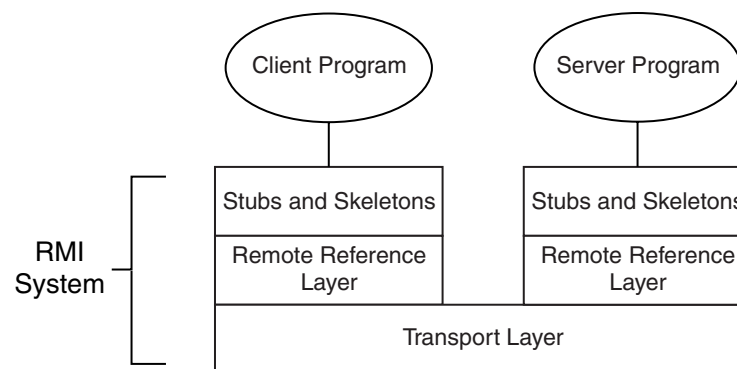
RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, supporting object-oriented polymorphism. The RMI registry is a lookup service for ports.

The RMI implementation

The RMI implementation consists of three abstraction layers.

These abstraction layers are:

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



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

1. Objects that require remote behavior should extend the `RemoteObject` class, typically through the `UnicastRemoteObject` subclass.
 - a. The `UnicastRemoteObject` subclass exports the remote object to make it available for servicing incoming RMI calls.
 - b. Exporting the remote object creates a new server socket, which is bound to a port number.
 - c. A thread is also created that listens for connections on that socket. The server is registered with a registry.

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

Thread pooling for RMI connection handlers

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

Thread pooling is not enabled by default. Enable it with this command-line setting:
`-Dsun.rmi.transport.tcp.connectionPool=true`

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

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

Understanding distributed garbage collection

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

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

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

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

Debugging applications involving RMI

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

The list of exceptions that can occur when using RMI and their context is included in the *RMI Specification* document at: <http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmi-exceptions.html#3601>

Properties settings that are useful for tuning, logging, or tracing RMI servers and clients can be found at: <http://java.sun.com/j2se/1.5.0/docs/guide/rmi/javarmiproperties.html>

Solutions to some common problems and answers to frequently asked questions related to RMI and object serialization can be found at: <http://java.sun.com/j2se/1.5.0/docs/guide/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.

Chapter 7. The ORB

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

The topics in this chapter are:

- “Using the ORB” on page 51
- “How the ORB works” on page 54
- “Additional features of the ORB” on page 61
- “CORBA”
- “RMI and RMI-IIOP” on page 44
- “Java IDL or RMI-IIOP?” on page 44
- “RMI-IIOP limitations” on page 44
- “Further reading” on page 45
- “Examples of client-server applications” on page 45

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

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 Sun is to use Java IDL when you are using Java to access existing CORBA resources, and RMI-IIOP to export RMI resources to CORBA.

RMI-IIOP limitations

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

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

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

Further reading

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

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

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

You can find the RMI-IIOP Programmer's Guide in your SDK installation directory under `docs/common/rmi-iiop.html`. Example programs are provided in `demo/rmi-iiop`.

Examples of client-server applications

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

Interfaces

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

The two interfaces are:

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

```
interface Sample { string message(); };
```
- Java RMI Interface (`Sample.java`):

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

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

Remote object implementation (or servant)

This description shows possible implementations of the object.

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

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

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

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

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

The POA is the Portable Object Adapter, described in “Portable object adapter” on page 61.

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

Stubs and ties generation

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

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

CORBA	RMI(JRMP)	RMI-IIOP
<code>idlj Sample.idl</code>	<code>rmic SampleImpl</code>	<code>rmic -iiop Sample</code>

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

CORBA	RMI(JRMP)	RMI-IIOP
<code>Sample.java</code>	<code>SampleImpl_Skel.class</code>	<code>_SampleImpl_Tie.class</code>
<code>SampleHolder.java</code>	<code>SampleImpl_Stub.class</code>	<code>_Sample_Stub.class</code>
<code>SampleHelper.java</code>	<code>Sample.class</code> (<code>Sample.java</code> present)	<code>Sample.class</code> (<code>Sample.java</code> present)
<code>SampleOperations.java</code>	<code>SampleImpl.class</code> (only compiled)	<code>SampleImpl.class</code> (only compiled)
<code>_SampleStub.java</code>		
<code>SamplePOA.java</code> (-fserver , -fall , -fserverTie , -fallTie)		

CORBA	RMI(JRMP)	RMI-IIOP
SamplePOATie.java (-fserverTie, -fallTie)		
_SampleImplBase.java (-oldImplBase)		

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

Server code

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

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

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

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

java.naming.factory.initial

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

java.naming.provider.url

This property configures the root naming context, the Object Request Broker (ORB), or both. It is used when the naming service is stored in a different host, and it can take several URI schemes:

- rmi
- corbaname
- corbaloc
- IOR
- iiop
- iiopname

For example:

```

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

```

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

```

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

```

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

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

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

```

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

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

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

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

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

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

```

For RMI-IIOP:

```

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

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

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

```



```

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

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

```

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

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

RMI Client example:

```

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

```

CORBA Client example:

```

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

```

RMI-IIOP Client example:

```

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

```

```

        // Method Invocation
        System.out.println(sampleRef.message());
    }
    catch(Exception e) {}
}
}

```

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

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

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

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

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

Summary of differences in server development

There are a number of differences in server development.

- Import statement:

```
import javax.rmi.PortableRemoteObject;
```
- Implementation class of a remote object:

```
public class SampleImpl extends PortableRemoteObject implements Sample
```
- Name registration of a remote object:

```
NamingContext.rebind("Sample",ObjRef);
```
- Generate a tie for IIOP using the command:

```
rmic -iiop
```
- Run `tnameserv` as a name server.
- Generate IDL for CORBA clients using the command:

```
rmic -idl
```

Summary of differences in client development

There are a number of differences in client development.

- Import statement:

```
import javax.rmi.PortableRemoteObject;
```
- Identify a remote object by name:

```
Object obj = ctx.lookup("Sample")

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

```
rmic -iiop
```

Using the ORB

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

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

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

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

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

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

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

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

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

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

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

- For applications: `InetAddress.getLocalHost().getHostAddress()`
- For applets: `<applet>.getCodeBase().getHost()`

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

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

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

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

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

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

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

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

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

- **com.ibm.CORBA.enableLocateRequest:** (default: false)
If this property is set, the ORB sends a LocateRequest before the actual Request.
- **com.ibm.CORBA.FragmentSize:** (range: 0 through 2147483647=Java max int) (default:1024)
Controls GIOP 1.2 fragmentation. The size specified is rounded down to the nearest multiple of 8, with a minimum size of 64 bytes. You can disable message fragmentation by setting the value to 0.
- **com.ibm.CORBA.FragmentTimeout:** (range: 0 through 600000 ms) (default: 300000)
The maximum length of time for which the ORB waits for second and subsequent message fragments before timing out. Set this property to 0 if timeout is not required.
- **com.ibm.CORBA.GIOPAddressingDisposition:** (range: 0 through 2) (default: 0)
When a GIOP 1.2 Request, LocateRequest, Reply, or LocateReply is created, the addressing disposition is set depending on the value of this property:
 - 0 = Object Key
 - 1 = GIOP Profile
 - 2 = full IOR
 If this property is not set or is passed an invalid value, the default 0 is used.
- **com.ibm.CORBA.InitialReferencesURL:**
The format of the value of this property is a correctly formed URL; for example, `http://w3.mycorp.com/InitRefs.file`. The actual file contains a name and value pair like: `NameService=<stringified_IOR>`. If you specify this property, the ORB does not attempt the bootstrap approach. Use this property if you do not have a bootstrap server and want to have a file on the webserver that serves the purpose.

Note: This property is deprecated.
- **com.ibm.CORBA.ListenerPort:** (range: 0 through 2147483647=Java max int) (default: next available system assigned port number)
The port on which this server listens for incoming requests. If this property is specified, the ORB starts to listen during `ORB.init()`.
- **com.ibm.CORBA.LocalHost:**
The value of this property is a string. This string can be a host name or the IP address (ex. 9.5.88.112). If this property is not set, retrieve the local host by calling: `InetAddress.getLocalHost().getHostAddress()`. This property represents the host name (or IP address) of the system on which the ORB is running. The local host name is used by the server-side ORB to place the host name of the server into the IOR of a remote-able object.
- **com.ibm.CORBA.LocateRequestTimeout:** (range: 0 through 2147483647) (default: 0=infinity)
Defines the number of seconds to wait before timing out on a LocateRequest message.

- **com.ibm.CORBA.MaxOpenConnections:** (range: 0 through 2147483647) (default: 240)
Determines the maximum number of in-use connections that are to be kept in the connection cache table at any one time.
- **com.ibm.CORBA.MinOpenConnections:** (range: 0 through 2147483647) (default: 100)
The ORB cleans up only connections that are not busy from the connection cache table, if the size of the table is higher than the MinOpenConnections.
- **com.ibm.CORBA.NoLocalInterceptors:** (default: false)
If this property is set to true, no local portable interceptors are used. The expected result is improved performance if interceptors are not required when connecting to a co-located object.
- **com.ibm.CORBA.ORBCharEncoding:** (default: ISO8859_1)
Specifies the native encoding set used by the ORB for character data.
- **com.ibm.CORBA.ORBWCharDefault:** (default: UCS2)
Indicates that wchar code set UCS2 is to be used with other ORBs that do not publish a wchar code set.
- **com.ibm.CORBA.RequestTimeout:** (range: 0 through 2147483647) (default: 0=infinity)
Defines the number of seconds to wait before timing out on a Request message.
- **com.ibm.CORBA.SendingContextRunTimeSupported:** (default: true)
Set this property to false to disable the CodeBase SendingContext RunTime service. This means that the ORB does not attach a SendingContextRunTime service context to outgoing messages.
- **com.ibm.CORBA.SendVersionIdentifier:** (default: false)
Tells the ORB to send an initial dummy request before it starts to send any real requests to a remote server. This action determines the partner version of the remote server ORB, based on the response from that ORB.
- **com.ibm.CORBA.ServerSocketQueueDepth:** (range: 50 through 2147483647) (default: 0)
The maximum queue length for incoming connection indications. A connect indication is a request to connect. If a connection indication arrives when the queue is full, the connection is refused. If the property is not set, the default 0 is used. If the property is not valid, 50 is used.
- **com.ibm.CORBA.ShortExceptionDetails:** (default: false)
When a CORBA SystemException reply is created, the ORB, by default, includes the Java stack trace of the exception in an associated ExceptionDetailMessage service context. If you set this property to any value, the ORB includes a toString of the Exception instead.
- **com.ibm.tools.rmic.iiop.Debug:** (default: false)
The rmic tool automatically creates import statements in the classes that it generates. If set to true, this property causes rmic to report the mappings of fully qualified class names to short names.
- **com.ibm.tools.rmic.iiop.SkipImports:** (default: false)
If this property is set to true, classes are generated with rmic using fully qualified names only.
- **org.omg.CORBA.ORBId**
Uniquely identifies an ORB in its address space. For example, the address space might be the server containing the ORB. The ID can be any String. The default value is a randomly generated number that is unique in the JVM of the ORB.

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

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

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

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

- **org.omg.CORBA.ORBServerId**

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

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

Sun Microsystems Inc. property	IBM property
com.sun.CORBA.ORBServerHost	com.ibm.CORBA.LocalHost
com.sun.CORBA.ORBServerPort	com.ibm.CORBA.ListenerPort
org.omg.CORBA.ORBInitialHost	com.ibm.CORBA.BootstrapHost
org.omg.CORBA.ORBInitialPort	com.ibm.CORBA.BootstrapPort
org.omg.CORBA.ORBInitialServices	com.ibm.CORBA.InitialReferencesURL

How the ORB works

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

This section describes a basic, typical RMI-IIOP session in which a client accesses a remote object on a server. The access is made possible through an interface named `Sample`. The client calls a simple method provided through the interface. The method is called `message()`. The method returns a "Hello World" string. For further examples, see "Examples of client-server applications" on page 45.

The client side

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

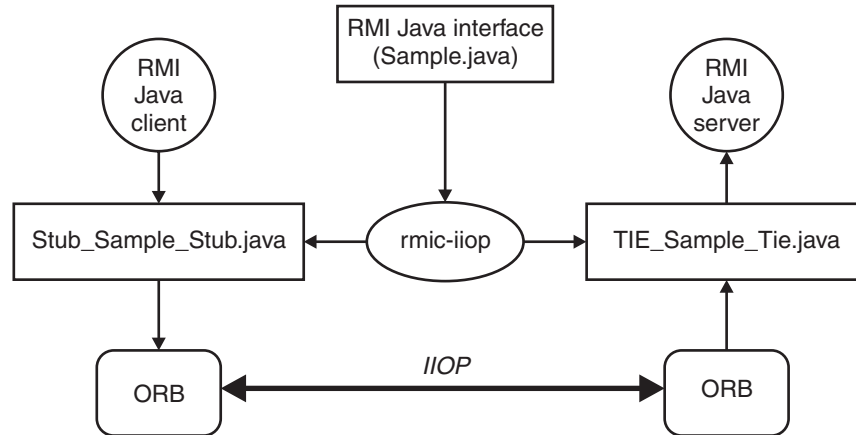
The subjects discussed here are:

- "Stub creation" on page 55
- "ORB initialization" on page 55
- "Obtaining the remote object" on page 56
- "Remote method invocation" on page 57

Stub creation

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

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



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

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

```
rmic -iiop Sample
```

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

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

ORB initialization

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

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

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

The parameters that are passed to the method are:

- A string array containing property-value pairs.
- A Java Properties object.

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

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

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

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

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

Obtaining the remote object

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

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

```
IOR:0000000000000001d524d493a5.....
```

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

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

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

In the example:

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

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

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

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

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

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

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

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

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

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

Remote method invocation

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

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

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

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

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

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

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

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

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

See Chapter 17, “ORB problem determination,” on page 193 for a detailed description and example.

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

A service context is some extra information that the ORB includes in the request or reply, to add several other functions. CORBA defines a few service contexts, such

as the codebase and the codeset service contexts. The first is used for the callback feature which is described in the CORBA specification. The second context is used to specify the encoding of strings.

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

The server side

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

The subjects discussed here are:

- “Servant implementation”
- “Tie generation”
- “Servant binding”
- “Processing a request” on page 60

Servant implementation

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

Tie generation

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

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

```
rmic -iiop SampleImpl
```

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

Servant binding

The steps required to bind the servant are described.

The server implementation is required to do the following tasks:

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

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

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

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

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

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

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

Processing a request

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

During the ORB initialization, a listener thread was created. The listener thread is listening on a default port (the next available port at the time the thread was created). You can specify the listener port by using the `com.ibm.CORBA.ListenerPort` property. When a request comes in through that port, the listener thread first creates a connection with the client side. In this case, it is the TCP transport layer that takes care of the details of the connection. The ORB caches all the connections that it creates.

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

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

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

Additional features of the ORB

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

This section describes:

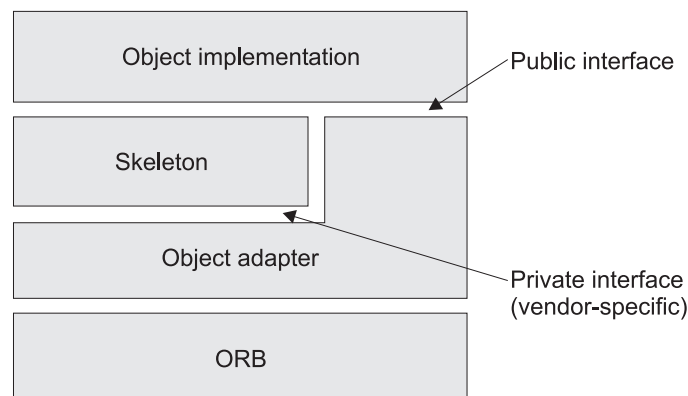
- “Portable object adapter”
- “Fragmentation” on page 63
- “Portable interceptors” on page 63
- “Interoperable Naming Service (INS)” on page 66

Portable object adapter

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

The main responsibilities of an object adapter are:

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



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

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

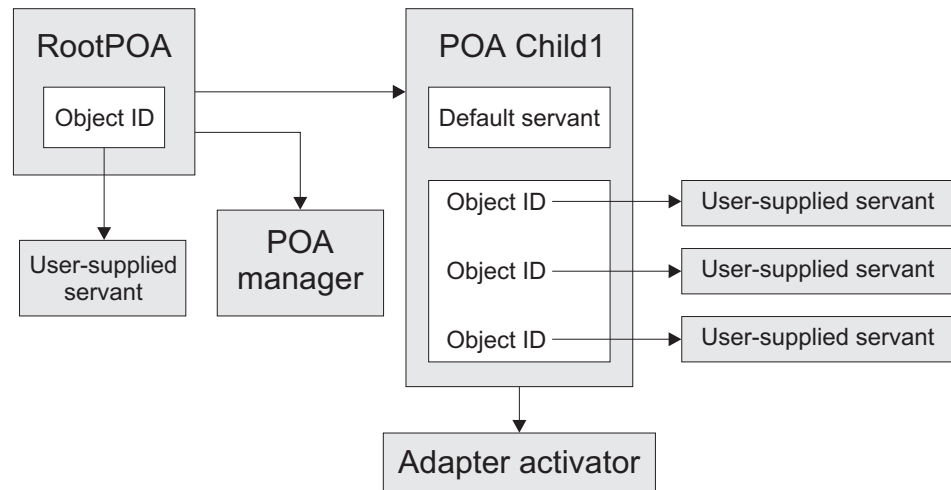
- Allow programmers to construct object and server implementations that are portable between different ORB products.
- Provide support for persistent objects. The support enables objects to persist across several server lifetimes.
- Support transparent activation of objects.
- Associate policy information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

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

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

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

Ultimately, a POA handles Object IDs and active servants. An active servant is a programming object that exists in memory. The servant is registered with the POA because one or more associated object identities was used. The ORB and POA cooperate to determine which servant starts the operation requested by the client. By using the POA APIs, you can create a reference for the object, associate an object ID, and activate the servant for that object. A map of object IDs and active servants is stored inside the POA. A POA also provides a default servant that is used when no active servant has been registered. You can register a particular implementation of this default servant. You can also register a servant manager, which is an object for managing the association of an object ID with a particular servant.



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

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

Fragmentation

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

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

Portable interceptors

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

CORBA implementations have mechanisms for users to insert their own code into the ORB processing flow. The code is inserted into the flow at “interception points”. The result is that the code, known as an interceptor, is called at particular stages during the processing of requests. It can directly inspect and even manipulate requests. Because this message filtering mechanism is flexible and powerful, the OMG standardized interceptors in the CORBA 2.4.2 specification under the name “portable interceptors”.

The idea of a portable interceptor is to define a standard interface. The interface enables you to register and run application-independent code that, among other things, takes care of passing service contexts. These interfaces are stored in the

package org.omg.PortableInterceptor.*. The implementation classes are in the com.ibm.rmi.pi.* package of the IBM ORB. All the interceptors implement the Interceptor interface.

Two classes of interceptors are defined:

Request interceptors

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

Interoperable Object Reference (IOR) interceptors

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

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

Five interception points are available on the client side:

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

Five interception points are available on the server side:

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

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

A simple interceptor is shown in the following example:

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

    public void destroy() {}

    // ClientRequestInterceptor operations
    public void send_request(ClientRequestInfo ri) {
        logger(ri, "send_request");
    }

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



```

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

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

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

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

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

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

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

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

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

```

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

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

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

```

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

    public String name() {
        return "";
    }

    public void pre_init(ORBInitInfo info) {

```

```

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

public void post_init(ORBInitInfo info) {}
}

```

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

```

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

```

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

Interoperable Naming Service (INS)

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

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

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

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

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

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

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

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

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

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

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

An example corbaname URI is:

```
corbaname::myOrg.com:2050#Personal/schedule
```

In this example, the portion of the reference up to the number sign character “#” is the URL that returns the root naming context. The second part of the example, after the number sign character “#”, is the argument that is used to resolve the object on the NamingContext.

The INS specified two standard command-line arguments that provide a portable way of configuring ORB::resolve_initial_references():

- **-ORBInitRef** takes an argument of the form <ObjectId>=<ObjectURI>. For example, you can use the following command-line arguments:

```
-ORBInitRef NameService=corbaname::myhost.example.com
```

In this example, resolve_initial_references("NameService") returns a reference to the object with key NameService available on myhost.example.com, port 2809.

- **-ORBDefaultInitRef** provides a prefix string that is used to resolve otherwise unknown names. When resolve_initial_references() cannot resolve a name that has been configured with **-ORBInitRef**, it constructs a string that consists of the default prefix, a “/” character, and the name requested. The string is then supplied to string_to_object(). For example, with a command line of:

```
-ORBDefaultInitRef corbaloc::myhost.example.com
```

a call to resolve_initial_references("MyService") returns the object reference that is denoted by corbaloc::myhost.example.com/MyService.

Chapter 8. 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 Sun Microsystems Inc. IBM recommends that you read the JNI specification. Go to <http://java.sun.com/> and search the site for JNI. Sun Microsystems maintain a combined programming guide and specification at <http://java.sun.com/docs/books/jni/>.

This section gives additional information to help you with JNI operation and design.

The topics that are discussed in this section are:

- “Overview of JNI”
- “The JNI and the Garbage Collector” on page 70
- “Copying and pinning” on page 75
- “Handling exceptions” on page 76
- “Synchronization” on page 77
- “Debugging the JNI” on page 78
- “JNI checklist” on page 79

Overview of JNI

From the viewpoint of a JVM, there are two types of code: “Java” and “native”. The Java Native Interface (JNI) establishes a well-defined and platform-independent interface between the two.

Native code can be used together with Java in two distinct ways: as “native methods” in a running JVM and as the code that creates a JVM using the “Invocation API”. This section describes the difference.

Native methods

Java native methods are declared in Java, implemented in another language (such as C or C++), and loaded by the JVM as necessary. To use native methods, you must:

1. **Declare** the native method in your Java code.

When the `javac` compiler encounters a native method declaration in Java source code, it records the name and parameters for the method. Because the Java source code contains no implementation, the compiler marks the method as “native”. The JVM can then resolve the method correctly when it is called.

2. **Implement** the native method.

Native methods are implemented as external entry points in a loadable binary library. The contents of a native library are platform-specific. The JNI provides a way for the JVM to use any native methods in a platform-independent way.

The JVM performs calls to native methods. When the JVM is in a native method, JNI provides a way to “call back” to the JVM.

3. **Load** the native method code for the VM to use.

As well as declaring the native method, you must find and load the native library that contains the method at runtime.

Two Java interfaces load native libraries:

- `java.lang.System.load()`
- `java.lang.System.loadLibrary()`

Typically, a class that declares native methods loads the native library in its static initializer.

Invocation API

Creating a JVM involves native code. The aspect of the JNI used for this purpose is called the JNI Invocation API. To use the Invocation API, you bind to an implementation-specific shared library, either statically or dynamically, and call the `JNI_*` functions it exports.

The JNI specification and implementation

The JNI specification is vague on selected implementation details. It provides a reusable framework for simple and extensible C and C++ native interfaces. The JNI model is also the basis for the JVMTI specification.

The Sun Microsystems 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 Chapter 2, “Memory management,” on page 7.

To collect unreachable objects, the GC must know when Java objects are referenced by native code. The JNI implementation uses “root sets” to ensure that objects can be reached. A root set is a set of direct, typically relocatable, object references that are traceable by the GC.

There are several types of root set. The union of all root sets provides the starting set of objects for a GC mark phase. Beginning with this starting set, the GC traverses the entire object reference graph. Anything that remains unmarked is unreachable garbage. (This description is an over-simplification when reachability and weak references are considered. See “Detailed description of garbage collection” on page 11 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 76.

To provide JNI global reference capabilities and also provide some automatic garbage collection of the referents, the JNI provides two functions:

- NewWeakGlobalRef
- DeleteWeakGlobalRef

These functions provide JNI access to weak references.

Local references and memory leaks

The automatic garbage collection of local references that are no longer in scope prevents memory leaks in most situations. This automatic garbage collection occurs when a native thread returns to Java (native methods) or detaches from the JVM (Invocation API). Local reference memory leaks are possible if automatic garbage collection does not occur. A memory leak might occur if a native method does not return to the JVM, or if a program that uses the Invocation API does not detach from the JVM.

Consider the code in the following example, where native code creates new local references in a loop:

```
while ( <condition> )
{
    jobject myObj = (*env)->NewObject( env, clz, mid, NULL );

    if ( NULL != myObj )
    {
        /* we know myObj is a valid local ref, so use it */
        jclass myClazz = (*env)->GetObjectClass(env, myObj);

        /* uses of myObj and myClazz, etc. but no new local refs */

        /* Without the following calls, we would leak */
        (*env)->DeleteLocalRef( env, myObj );
        (*env)->DeleteLocalRef( env, myClazz );
    }
}

} /* end while */
```

Although new local references overwrite the myObj and myClazz variables inside the loop, every local reference is kept in the root set. These references must be explicitly removed by the DeleteLocalRef call. Without the DeleteLocalRef calls, the

local references are leaked until the thread returned to Java or detached from the JVM.

JNI weak global references

Weak global references are a special type of global reference. They can be used in any thread and can be used between native function calls, but do not act as GC roots. The GC disposes of an object that is referred to by a weak global reference at any time if the object does not have a strong reference elsewhere.

You must use weak global references with caution. If the object referred to by a weak global reference is garbage collected, the reference becomes a null reference. A null reference can only safely be used with a subset of JNI functions. To test if a weak global reference has been collected, use the `IsSameObject` JNI function to compare the weak global reference to the null value.

It is not safe to call most JNI functions with a weak global reference, even if you have tested that the reference is not null, because the weak global reference could become a null reference after it has been tested or even during the JNI function. Instead, a weak global reference should always be promoted to a strong reference before it is used. You can promote a weak global reference using the `NewLocalRef` or `NewGlobalRef` JNI functions.

Weak global references use memory and must be freed with the `DeleteWeakGlobalRef` JNI function when it is no longer needed. Failure to free weak global references causes a slow memory leak, eventually leading to out-of-memory exceptions.

For information and warnings about the use of JNI global weak references, see the JNI specification.

JNI reference management

There are a set of platform-independent rules for JNI reference management

These rules are:

1. JNI references are valid only in threads attached to a JVM.
2. A valid JNI local reference in native code must be obtained:
 - a. As a parameter to the native code
 - b. As the return value from calling a JNI function
3. A valid JNI global reference must be obtained from another valid JNI reference (global or local) by calling `NewGlobalRef` or `NewWeakGlobalRef`.
4. The null value reference is always valid, and can be used in place of any JNI reference (global or local).
5. JNI local references are valid only in the thread that creates them and remain valid only while their creating frame remains on the stack.

Note:

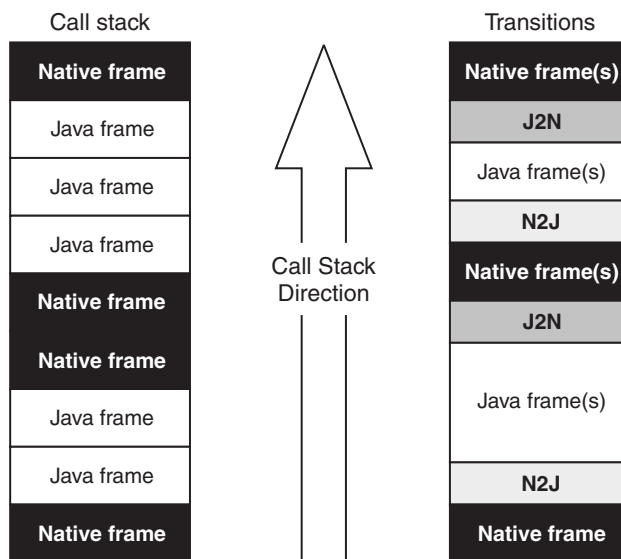
1. Overwriting a local or global reference in native storage with a null value does not remove the reference from the root set. Use the appropriate `Delete*Ref` JNI function to remove references from root sets.
2. Many JNI functions (such as `FindClass` and `NewObject`) return a null value if there is an exception pending. Comparing the returned value to the null value

for these calls is semantically equivalent to calling the JNI `ExceptionCheck` function. See the JNI specification for more details.

3. A JNI local reference must never be used after its creating frame returns, regardless of the circumstances. It is dangerous to store a JNI local reference in any process static storage.

JNI transitions

To understand JNI local reference management and the GC, you must understand the context of a running thread attached to the JVM. Every thread has a runtime stack that includes a frame for each method call. From a GC perspective, every stack establishes a thread-specific "root set" including the union of all JNI local references in the stack.



Each method call in a running VM adds (pushes) a frame onto the stack, just as every return removes (pops) a frame. Each call point in a running stack can be characterized as one of the following:

- Java to Java (J2J)
- Native to Native (N2N)
- Java to Native (J2N)
- Native to Java (N2J)

You can only perform an N2J transition in a thread that meets the following conditions:

- The process containing the thread must contain a JVM started using the JNI Invocation API.
- The thread must be "attached" to the JVM.
- The thread must pass at least one valid local or global object reference to JNI.

J2J and N2N transitions

Because object references do not change form as part of J2J or N2N transitions, J2J and N2N transitions do not affect JNI local reference management.

Any section of N2N code that obtains many local references without promptly returning to Java can needlessly stress the local reference capacity of a thread. This problem can be avoided if local references are managed explicitly by the native method programmer.

N2J transitions

For native code to call Java code (N2J) in the current thread, the thread must first be attached to the JVM in the current process.

Every N2J call that passes object references must have obtained them using JNI, therefore they are either valid local or global JNI refs. Any object references returned from the call are JNI local references.

J2N calls

The JVM must ensure that objects passed as parameters from Java to the native method and any new objects created by the native code remain reachable by the GC. To handle the GC requirements, the JVM allocates a small region of specialized storage called a "local reference root set".

A local reference root set is created when:

- A thread is first attached to the JVM (the "outermost" root set of the thread).
- Each J2N transition occurs.

The JVM initializes the root set created for a J2N transition with:

- A local reference to the caller's object or class.
- A local reference to each object passed as a parameter to the native method.

New local references created in native code are added to this J2N root set, unless you create a new "local frame" using the `PushLocalFrame` JNI function.

The default root set is large enough to contain 16 local references per J2N transition. The `-Xcheck:jni` command-line option causes the JVM to monitor JNI usage. When `-Xcheck:jni` is used, the JVM writes a warning message when more than 16 local references are required at runtime. If you receive this warning message, use one of the following JNI functions to manage local references more explicitly:

- `NewLocalRef`
- `DeleteLocalRef`
- `PushLocalFrame`
- `PopLocalFrame`
- `EnsureLocalCapacity`

J2N returns

When native code returns to Java, the associated JNI local reference "root set", created by the J2N call, is released.

If the JNI local reference was the only reference to an object, the object is no longer reachable and can be considered for garbage collection. Garbage collection is triggered automatically by this condition, which simplifies memory management for the JNI programmer.

Copying and pinning

The GC might, at any time, decide it needs to compact the garbage-collected heap. Compaction involves physically moving objects from one address to another. These objects might be referred to by a JNI local or global reference. To allow compaction to occur safely, JNI references are not direct pointers to the heap. At least one level of indirection isolates the native code from object movement.

If a native method needs to obtain direct addressability to the inside of an object, the situation is more complicated. The requirement to directly address, or pin, the heap is typical where there is a need for fast, shared access to large primitive arrays. An example might include a screen buffer. In these cases a JNI critical section can be used, which imposes additional requirements on the programmer, as specified in the JNI description for these functions. See the JNI specification for details.

- `GetPrimitiveArrayCritical` returns the direct heap address of a Java array, disabling garbage collection until the corresponding `ReleasePrimitiveArrayCritical` is called.
- `GetStringCritical` returns the direct heap address of a `java.lang.String` instance, disabling garbage collection until `ReleaseStringCritical` is called.

All other `Get<PrimitiveType>ArrayElements` interfaces return a copy that is unaffected by compaction.

Using the `isCopy` flag

The JNI `Get<Type>` functions specify a pass-by-reference output parameter (`jboolean *isCopy`) that allows the caller to determine whether a given JNI call is returning the address of a copy or the address of the pinned object in the heap.

The `Get<Type>` and `Release<Type>` functions come in pairs:

- `GetStringChars` and `ReleaseStringChars`
- `GetStringCritical` and `ReleaseStringCritical`
- `GetStringUTFChars` and `ReleaseStringUTFChars`
- `Get<PrimitiveType>ArrayElements` and `Release<PrimitiveType>ArrayElements`
- `GetPrimitiveArrayCritical` and `ReleasePrimitiveArrayCritical`

If you pass a non-null address as the `isCopy` parameter, the JNI function sets the `jboolean` value at that address to `JNI_TRUE` if the address returned is the address of a copy of the array elements and `JNI_FALSE` if the address points directly into the pinned object in the heap.

Except for the critical functions, the IBM JVM always returns a copy. Copying eases the burden on the GC, because pinned objects cannot be compacted and complicate defragmentation.

To avoid leaks, you must:

- Manage the copy memory yourself using the `Get<Type>Region` and `Set<Type>Region` functions.
- Ensure that you free copies made by a `Get<Type>` function by calling the corresponding `Release<Type>` function when the copy is no longer needed.

Using the mode flag

When you call `Release<Type>ArrayElements`, the last parameter is a mode flag. The mode flag is used to avoid unnecessary copying to the Java heap when working with a copied array. The mode flag is ignored if you are working with an array that has been pinned.

You must call `Release<Type>` once for every `Get<Type>` call, regardless of the value of the `isCopy` parameter. This step is necessary because calling `Release<Type>` deletes JNI local references that might otherwise prevent garbage collection.

The possible settings of the mode flag are:

0 Update the data on the Java heap. Free the space used by the copy.

JNI_COMMIT

Update the data on the Java heap. **Do not** free the space used by the copy.

JNI_ABORT

Do not update the data on the Java heap. Free the space used by the copy.

The '0' mode flag is the safest choice for the `Release<Type>` call. Whether the copy of the data was changed or not, the heap is updated with the copy, and there are no leaks.

To avoid having to copy back an unchanged copy, use the `JNI_ABORT` mode value. If you alter the returned array, check the `isCopy` flag before using the `JNI_ABORT` mode value to "roll back" changes. This step is necessary because a pinning JVM leaves the heap in a different state than a copying JVM.

A generic way to use the `isCopy` and mode flags

Here is a generic way to use the `isCopy` and mode flags. It works with all JVMs and ensures that changes are committed and leaks do not occur.

To use the flags in a generic way, ensure that you:

- Do not use the `isCopy` flag. Pass in null or 0.
- Always set the mode flag to zero.

A complicated use of these flags is necessary only for optimization. If you use the generic way, you must still consider synchronization. See "Synchronization" on page 77.

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 bottom of this section).

pedantic

Perform more thorough, but slower checks.

valist

Check for va_list reuse (see the note at the bottom of the section).

help

Print help information.

The **-Xcheck:jni** option might reduce performance because it is thorough when it validates the supplied parameters.

Note:

On some platforms, reusing a va_list in a second JNI call (for example, when calling CallStaticVoidMethod() twice with the same arguments) causes the va_list to be corrupted and the second call to fail. To ensure that the va_list is not corrupted, use the standard C macro va_copy() in the first call. By default, **-Xcheck:jni** ensures that va_lists are not being reused. Use the **novalist** suboption to disable this check only if your platform allows reusing va_list without va_copy. z/OS platforms allow va_list reuse, and by default **-Xcheck:jni:novalist** is used. To enable va_list reuse checking, use the **-Xcheck:jni:valist** option.

JNI checklist

There are a number of items that you must remember when using the JNI.

The following table shows the JNI checklist:

Remember	Outcome of nonadherence
Local references cannot be saved in global variables.	Random crashes (depending on what you pick up in the overwritten object space) happen at random intervals.
Ensure that every global reference created has a path that deletes that global reference.	Memory leak. It might throw a native exception if the global reference storage overflows. It can be difficult to isolate.
Always check for exceptions (or return codes) on return from a JNI function. Always handle a deferred exception immediately you detect it.	Unexplained exceptions or undefined behavior. Might crash the JVM.
Ensure that array and string elements are always freed.	A small memory leak. It might fragment the heap and cause other problems to occur first.
Ensure that you use the isCopy and mode flags correctly. See “A generic way to use the isCopy and mode flags” on page 76.	Memory leaks, heap fragmentation, or both.

Remember	Outcome of nonadherence
When you update a Java object in native code, ensure synchronization of access.	Memory corruption.

Part 2. Submitting problem reports

If you find a problem with Java, make a report through the product that supplied the Java SDK, or through the Operating System if there is no bundling product.

On z/OS, the Java SDK is bundled with WebSphere Application Server only. It is not included with the Operating System, but is made available as a separate download. If you have a problem using Java on z/OS, submit a problem report through support for the product that is using the Java SDK.

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.

Part 3. 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:

- Chapter 9, "First steps in problem determination," on page 85
- Chapter 10, "AIX problem determination," on page 87
- Chapter 11, "Linux problem determination," on page 121
- Chapter 12, "Windows problem determination," on page 139
- Chapter 13, "z/OS problem determination," on page 149
- Chapter 14, "IBM i problem determination," on page 167
- Chapter 15, "Sun Solaris problem determination," on page 189
- Chapter 16, "Hewlett-Packard SDK problem determination," on page 191
- Chapter 17, "ORB problem determination," on page 193
- Chapter 18, "NLS problem determination," on page 207

Chapter 9. First steps in problem determination

Before proceeding in problem determination, there are some initial questions to be answered.

Have you changed anything recently?

If you have changed, added, or removed software or hardware just before the problem occurred, back out the change and see if the problem persists.

What else is running on the workstation?

If you have other software, including a firewall, try switching it off to see if the problem persists.

Is the problem reproducible on the same workstation?

Knowing that this defect occurs every time the described steps are taken is helpful because it indicates a straightforward programming error. If the problem occurs at alternate times, or occasionally, thread interaction and timing problems in general are much more likely.

Is the problem reproducible on another workstation?

A problem that is not evident on another workstation might help you find the cause. A difference in hardware might make the problem disappear; for example, the number of processors. Also, differences in the operating system and application software installed might make a difference to the JVM. For example, the visibility of a race condition in the JVM or a user Java application might be influenced by the speed at which certain operations are performed by the system.

Does the problem occur on multiple platforms?

If the problem occurs only on one platform, it might be related to a platform-specific part of the JVM. Alternatively, it might be related to local code used inside a user application. If the problem occurs on multiple platforms, the problem might be related to the user Java application. Alternatively, it might be related to a cross-platform part of the JVM such as the Java Swing API. Some problems might be evident only on particular hardware; for example, Intel® 32 bit architecture. A problem on particular hardware might indicate a JIT problem.

Can you reproduce the problem with the latest Service Refresh?

The problem might also have been fixed in a recent service refresh. Make sure that you are using the latest service refresh for your environment. Check the latest details on <http://www.ibm.com/developerWorks>.

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 Part 3, "Problem determination," on page 83 section, and check for latest details on the Developer Works Web site <http://www.ibm.com/developerWorks>.

Does turning off the JIT help?

If turning off the JIT prevents the problem, there might be a problem with the JIT. 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 Chapter 26, “JIT problem determination,” on page 317.)

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 Chapter 21, “Using dump agents,” on page 223 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?

Information about any problems is produced by the JVM. You can enable more detailed logging, and control where the logging information goes. For more details, see Appendix C, “Messages,” on page 419.

Chapter 10. AIX problem determination

This section describes problem determination on AIX.

The topics are:

- “Setting up and checking your AIX environment”
- “General debugging techniques” on page 89
- “Diagnosing crashes” on page 100
- “Debugging hangs” on page 102
- “Understanding memory usage” on page 105
- “Debugging performance problems” on page 113
- “MustGather information for AIX” on page 119

Setting up and checking your AIX environment

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

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

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

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

The default installation directory is in `/usr/java5` for the 32-bit JVM and `/usr/java5_64` `/usr/java6_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/java5/jre/bin:/usr/java5/bin` in your **PATH** environment variable. If it is not present, add it by using `export PATH=/usr/java5/jre/bin:/usr/java5/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 install using `smitty` or `installp`. You can find the list of prerequisites in the *User Guide* that is shipped with the SDK. Use `lslpp -l` to find the list of current filesets. Use `instfix -i -k <apar number>` to test for the presence of an APAR and `instfix -i | grep _ML` to find the installed maintenance level.

The **ReportEnv** tool, available from the Java service team, plugs into your JVM and reports on the JVM environment in real time. Your JVM environment affects the operation of the JVM. ReportEnv reports on environment variables and command-line parameters. It is a GUI tool, although it can be run without a GUI. The GUI allows you to browse your environment and, to some extent, dynamically change it. The tool also has a mechanism to generate reports to tell you the exact state of your JVM environment. A screen capture of the tool is shown in Figure 1 on page 140. The ReportEnv tool is available on request from jvmcookbook@uk.ibm.com.

Directory requirements

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

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

The system dump agents can also be configured on the command line. See Chapter 21, “Using dump agents,” on page 223 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:

<code>ulimit -c unlimited</code>	turn on corefiles with unlimited size
<code>ulimit -n unlimited</code>	allows an unlimited number of open file descriptors
<code>ulimit -d unlimited</code>	sets the user data limit to unlimited
<code>ulimit -f unlimited</code>	sets the file limit to unlimited

You can display the current `ulimit` settings with:

```
ulimit -a
```

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

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

2. Set the following in smitty:
 - a. Start smitty as root

- b. Go to **System Environments** → **Change/Show Characteristics of Operating System**
- c. Set the **Enable full CORE dump** option to TRUE
- d. Ensure that the **Use pre-430 style CORE dump** option is set to FALSE

Alternatively, you can run:

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

Java Virtual Machine settings

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

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

```
java -Xdump:what
```

which should include something like the following:

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

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

You can change and set options using the command-line option **-Xdump**, which is described in Chapter 21, “Using dump agents,” on page 223.

Available disk space

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

General debugging techniques

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

In addition to this information, you can obtain AIX publications from the IBM System p® and AIX Information Center: <http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp>. Of particular interest are:

- Performance management and tuning
- Programming for AIX

You might also find *Developing and Porting C and C++ Applications on AIX* (SG24-5674) helpful, available from: <http://www.redbooks.ibm.com>.

There are several diagnostic tools available with the JVM to help diagnose problems:

- Starting Javadumps, see Chapter 22, “Using Javacore,” on page 245.
- Starting Heapdumps, see Chapter 23, “Using Heapdump,” on page 257.

- Starting system dumps, see Chapter 24, “Using system dumps and the dump viewer,” on page 263.

AIX provides various commands and tools that can be useful in diagnosing problems.

AIX debugging commands

List of debugging commands.

bindprocessor -q

Lists the available processors.

bootinfo -K

Shows if the 64-bit kernel is active.

bootinfo -y

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

dbx

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

The Java 5.0 SDK also includes a dbx Plug-in for additional help debugging Java applications. See “DBX Plug-in” on page 99 for more information.

iostat

Reports the read and write rate to all disks. This tool can help determine if disk workload should be spread across multiple disks. **iostat** also reports the same CPU activity that **vmstat** does.

lsattr

Details characteristics and values for devices in the system.

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

```
# lsattr -El proc0
state      enable           Processor state False
type       PowerPC_POWER3 Processor type   False
frequency  2000000000           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 91 for an example.

netpmon

uses the **trace** facility to obtain a detailed picture of network activity during a time interval. See “netpmon” on page 93 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 94 for more details.

nmon

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

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

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

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

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

ps Shows process information. See “ps” on page 94 for more details.

sar

Shows usage by multiple CPUs. See “sar” on page 96 for more details.

svmon

Captures snapshots of virtual memory. See “svmon” on page 96 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 98 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 98 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 99 for more details.

lsconf

This command shows basic hardware and configuration details.

For example:

```
System Model: IBM,7040-681
Machine Serial Number: 835A7AA
Processor Type: PowerPC_POWER4
Number Of Processors: 8
Processor Clock Speed: 1100 MHz
CPU Type: 64-bit
Kernel Type: 64-bit
LPAR Info: 5 JAVADEV1 - kukicha
Memory Size: 10240 MB
```

Good Memory Size: 10240 MB
 Platform Firmware level: 3H041021
 Firmware Version: IBM,RG041021_d78e05_s
 Console Login: enable
 Auto Restart: true
 Full Core: true

Network Information

Host Name: bblp5-1.hursley.ibm.com
 IP Address: 9.20.136.92
 Sub Netmask: 255.255.255.128
 Gateway: 9.20.136.1
 Name Server: 9.20.136.11
 Domain Name: hursley.ibm.com

Paging Space Information

Total Paging Space: 512MB
 Percent Used: 21%

Volume Groups Information

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

INSTALLED RESOURCE LIST

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

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

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

```
+ proc18          U1.18-P1-C4    Processor
+ proc22          U1.18-P1-C4    Processor
+ proc23          U1.18-P1-C4    Processor
```

netpmon

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

It also displays process CPU statistics that show:

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

For example,

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

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

Process CPU Usage Statistics:

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

The adapter usage is shown here:

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

The following example shows the java extract:

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

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

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

The following extract shows the thread output:

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

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

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

netstat

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

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

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

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

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

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

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

ps

Shows process information.

The Process Status (`ps`) is used to monitor:

- A process.
- Whether the process is still consuming CPU cycles.
- Which threads of a process are still running.

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

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

Your output should be:

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

Where

UID

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

PPID

The Parent Process ID.

PID

The Process ID.

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

STIME

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

TTY

The controlling workstation for the process.

TIME

The total execution time for the process.

CMD

The full command name and its parameters.

To see which threads are still running, type:

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

Your output should be:

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

Where

USER

The user name of the person running the process.

TID

The Kernel Thread ID of each thread.

ST

The state of the thread:

- O** Nonexistent.
- R** Running.
- S** Sleeping.
- W** Swapped.
- Z** Canceled.

T Stopped.

CP

CPU utilization of the thread.

PRI

Priority of the thread.

SC

Suspend count.

ARCHON

Wait channel.

F Flags.

TAT

Controlling terminal.

BAND

CPU to which thread is bound.

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

sar

Use the **sar** command to check the balance of processor usage for multiple processors.

In this following example, two samples are taken every 5 seconds on a twin-processor system that is running at 80% utilization.

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

```
AIX aix4prt 0 5 000544144C00 02/09/01
```

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

svmon

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

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

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

Gives output like:

Pid	Command	Inuse	Pin	Pgsp	Virtual	64-bit	Mthrd		
25084	AppS	78907	1570	182	67840	N	Y		
Vsid	Esid	Type	Description	Inuse	Pin	Pgsp	Virtual	Addr	Range
2c7ea	3	work	shmat/mmap	36678	0	0	36656	0..65513	
3c80e	4	work	shmat/mmap	7956	0	0	7956	0..65515	
5cd36	5	work	shmat/mmap	7946	0	0	7946	0..65517	

14e04	6 work shmat/mmap	7151	0	0	7151	0..65519
7001c	d work shared library text	6781	0	0	736	0..65535
0	0 work kernel seg	4218	1552	182	3602	0..22017 : 65474..65535
6cb5a	7 work shmat/mmap	2157	0	0	2157	0..65461
48733	c work shmat/mmap	1244	0	0	1244	0..1243
cac3	- pers /dev/hd2:176297	1159	0	-	-	0..1158
54bb5	- pers /dev/hd2:176307	473	0	-	-	0..472
78b9e	- pers /dev/hd2:176301	454	0	-	-	0..453
58bb6	- pers /dev/hd2:176308	254	0	-	-	0..253
cee2	- work	246	17	0	246	0..49746
4cbb3	- pers /dev/hd2:176305	226	0	-	-	0..225
7881e	- pers /dev/e2axa702-1:2048	186	0	-	-	0..1856
68f5b	- pers /dev/e2axa702-1:2048	185	0	-	-	0..1847
28b8a	- pers /dev/hd2:176299	119	0	-	-	0..118
108c4	- pers /dev/e2axa702-1:1843	109	0	-	-	0..1087
24b68	f work shared library data	97	0	0	78	0..1470
64bb9	- pers /dev/hd2:176311	93	0	-	-	0..92
74bbd	- pers /dev/hd2:176315	68	0	-	-	0..67
3082d	2 work process private	68	1	0	68	65287..65535
10bc4	- pers /dev/hd2:176322	63	0	-	-	0..62
50815	1 pers code,/dev/hd2:210969	9	0	-	-	0..8
44bb1	- pers /dev/hd2:176303	7	0	-	-	0..6
7c83e	- pers /dev/e2axa702-1:2048	4	0	-	-	0..300
34a6c	a mmap mapped to sid 44ab0	0	0	-	-	
70b3d	8 mmap mapped to sid 1c866	0	0	-	-	
5cb36	b mmap mapped to sid 7cb5e	0	0	-	-	
58b37	9 mmap mapped to sid 1cb66	0	0	-	-	
1c7c7	- pers /dev/hd2:243801	0	0	-	-	

in which:

Vsid

Segment ID

Esid

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

Type

Identifies the type of the segment:

pers Indicates a persistent segment.

work Indicates a working segment.

clnt Indicates a client segment.

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

Description

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

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

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

kernel

The segment is used by the kernel.

shared library

The segment is used for shared library text or data.

process private

Private data for the process.

shmat/mmap

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

Inuse

The number of pages in real memory from this segment.

Pin

The number of pages pinned from this segment.

Pgsp

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

Addr Range

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

topas

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

The screen looks like this:

```

Topas Monitor for host:   aix4prt
Mon Apr 16 16:16:50 2001 Interval: 2

Kernel  63.1  #####
User    36.8  #####
Wait     0.0
Idle     0.0

Network  KBPS  I-Pack  O-Pack  KB-In  KB-Out
lo0      213.9  2154.2  2153.7  107.0  106.9
tr0       34.7   16.9   34.4    0.9   33.8

Disk     Busy%  KBPS    TPS  KB-Read  KB-Writ
hdisk0   0.0    0.0    0.0    0.0     0.0

Name      PID  CPU%  PgSp  Owner
java     16684  83.6  35.1  root
java     12192  12.7  86.2  root
lrud      1032   2.7   0.0  root
aixterm   19502   0.5   0.7  root
topas     6908   0.5   0.8  root
ksh      18148   0.0   0.7  root
gil       1806   0.0   0.0  root

EVENTS/QUEUES  FILE/TTY
Cswitch    5984  Readch    4864
Syscall    15776  Writech   34280
Reads       8    Rawin      0
Writes     2469  Ttyout     0
Forks       0    Igets      0
Execs       0    Namei      4
Runqueue   11.5  Dirblk     0
Waitqueue   0.0

PAGING        MEMORY
Faults       3862  Real,MB    1023
Steals       1580  % Comp     27.0
PgspIn       0    % Noncomp  73.9
PgspOut      0    % Client   0.5
PageIn       0
PageOut      0  PAGING SPACE
Sios         0    Size,MB    512
              % Used     1.2
              % Free     98.7
NFS (calls/sec)
ServerV2     0
ClientV2     0  Press:
ServerV3     0  "h" for help

```

trace

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

While many of the other tools provide general statistics such as CPU and I/O utilization, the trace facility provides more detailed information. For example, you can find out:

- Where an event occurred in the code.
- Which process caused an event to occur.
- When an event took place.
- How an event is affecting the system.

The **curt** postprocessing tool can extract information from the trace. It provides statistics on CPU utilization and process and thread activity. Another postprocessing tool is **splat**, the Simple Performance Lock Analysis Tool. This tool can be used to analyze simple locks in the AIX kernel and kernel extensions.

vmstat

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

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

The general syntax of this command is:

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

A typical output looks like this:

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

In this output, look for:

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

DBX Plug-in

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

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

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

```
pluginload /usr/java5/jre/bin/libdbx_j9.so
```

You can also set the **DBX_PLUGIN_PATH** environment variable to `/usr/java5/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/java5/jre/bin/java
```

Under DBX, issue the command:

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

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

```
dbx -c .dbxinit
```

where `.dbxinit` is the default command file.

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

Diagnosing crashes

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

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

Documents to gather

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

- Collect the output generated by running `stackit.sh` against the core file. To find the core file, use **-Xdump:what** and look for the location shown in the `label` field. The command `stackit.sh` is a script that runs a dbx session and is available from your Support Representative or from jvmcookbook@uk.ibm.com.
- Collect the output generated by running `jextract` against the core file:

```
jextract [core file]
```

The format of the file produced is `core.{date}.{time}.{pid}.dmp.zip`. See Chapter 24, "Using system dumps and the dump viewer," on page 263 for details about `jextract`.

- Collect the javadump file. To find the javadump file, use **-Xdump:what** and look for the location shown in the label field.
- 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:

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

Locating the point of failure

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

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

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

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

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

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

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

```
general:
pthread addr = 0x302027e8      size      = 0x22c
vp addr      = 0x302057e4      size      = 0x294
thread errno = 0
start pc     = 0x10001120
joinable     = yes
pthread_t    = 1
scheduler:
kernel      =
```

```

        user          = 1 (other)
event :
  event              = 0x0
  cancel             = enabled, deferred, not pending
stack storage:
  base               = 0x2df23000
size                = 0x1fff7b0
  limit              = 0x2ff227b0
  sp                 = 0x2df2cc70

```

For the second cause, currently dbx (and therefore stackit.sh) does not understand the structure of the JIT and Interpreter stack frames, and is not capable of generating a stack trace from them. The javadump, however, does not suffer from this limitation and can be used to examine the stack trace. A failure in JIT-compiled code can be verified and examined using the JIT Debugging Guide (see Chapter 26, “JIT problem determination,” on page 317).

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 90. 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 252.

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 253. If you cannot diagnose the problem from the javadump and if the process still seems to be using processor cycles, either it has entered an infinite loop or it is suffering from very bad performance. Using **ps -mp [process id] -o THREAD** allows individual threads in a particular process to be monitored to determine which threads are using the CPU time. If the process has entered an infinite loop, it is likely that a small number of threads will be using the time. For example:

```
$ ps -mp 43824 -o THREAD
USER      PID  PPID      TID ST  CP  PRI  SC      WCHAN      F      TT  BND  COMMAND
wsuser    43824 51762      -  A   66   60  77      *    200001 pts/4  -   -   java ...
-         -    -      4021 S    0   60   1 22c4d670 c00400      -   -   -
-         -    -     11343 S    0   60   1 e6002cbc 8400400      -   -   -
-         -    -     14289 S    0   60   1 22c4d670 c00400      -   -   -
-         -    -     14379 S    0   60   1 22c4d670 c00400      -   -   -
...
-         -    -     43187 S    0   60   1 701e6114 400400      -   -   -
-         -    -     43939 R   33   76   1 20039c88 c00000      -   -   -
-         -    -     50275 S    0   60   1 22c4d670 c00400      -   -   -
-         -    -     52477 S    0   60   1 e600ccbc 8400400      -   -   -
...
-         -    -     98911 S    0   60   1 7023d46c 400400      -   -   -
-         -    -     99345 R   33   76   0      -    400000      -   -   -
-         -    -     99877 S    0   60   1 22c4d670 c00400      -   -   -
-         -    -    100661 S    0   60   1 22c4d670 c00400      -   -   -
-         -    -    102599 S    0   60   1 22c4d670 c00400      -   -   -
...
```

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

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

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

Now you can use **dbx** to generate the C thread stack for these two threads using the **dbx thread** command for the corresponding **dbx thread** numbers (*\$tx*). To obtain the full stack trace including Java frames, map the **dbx thread** number to the threads `pthread_t` value, which is listed by the Javdump file, and can be

obtained from the ExecEnv structure for each thread using the Dump Viewer. Do this with the **dbx** command thread info [dbx thread number], which produces an output of the form:

```

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

```

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

```

"Worker#31" (TID:0x36288b10, sys_thread_t:0x220c2db8) Native Thread State:
ThreadID: 00003233 Reuse: 1 USER SUSPENDED Native Stack Data : base: 22107f80
pointer 22106390 used(7152) free(250896)
----- Monitors held -----
java.io.OutputStreamWriter@3636a930
com.ibm.servlet.engine.webapp.BufferedWriter@3636be78
com.ibm.servlet.engine.webapp.WebAppRequestDispatcher@3636c270
com.ibm.servlet.engine.srt.SRTOutputStream@36941820
com.ibm.servlet.engine.oselister.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_oselister_nativeEntry_NativeServerConnection_nativeWrite

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

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

```


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

Poor performance on AIX

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

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

See “Debugging performance problems” on page 113 for more information about performance on AIX.

Understanding memory usage

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

32- and 64-bit JVMs

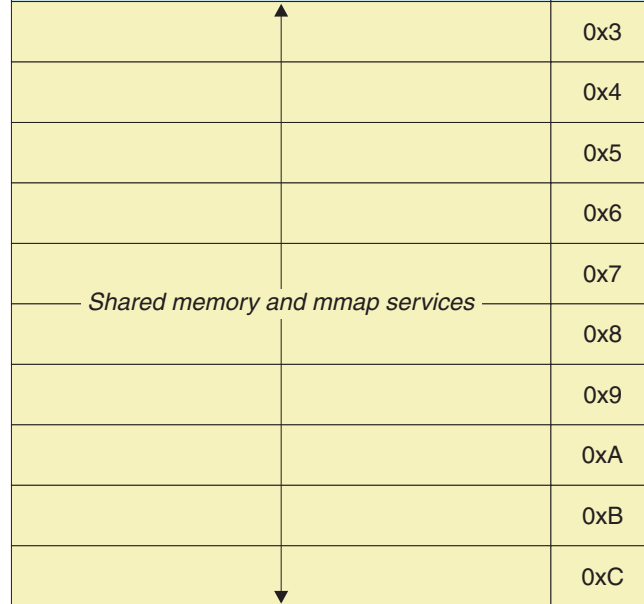
Most of the information in this section about altering the memory model and running out of native heap is relevant only to the 32-bit model, because the 64-bit model does not suffer from the same kind of memory constraints.

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

The 32-bit AIX Virtual Memory Model

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

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

Kernel	0x0
Application program text	0x1
Application program data and application stack	0x2
	0x3
	0x4
	0x5
	0x6
	0x7
	0x8
	0x9
	0xA
	0xB
	0xC
Shared library text	0xD
Miscellaneous kernel data	0xE
Application shared library data	0xF

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

The 64-bit AIX Virtual Memory Model

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

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

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

Changing the Memory Model (32-bit JVM)

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

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

The small memory model

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

The large memory model

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

You can alter the `o_maxdata` setting by:

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

The very large memory model

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

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

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

The native and Java heaps

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

The Java heap contains the instances of Java objects and is often referred to as 'the heap'. It is the Java heap that is maintained by Garbage Collection, and it is the Java heap that is changed by the command-line heap settings. The Java heap is allocated using mmap, or shmat if large page support is requested. The maximum size of the Java heap is preallocated during JVM startup as one contiguous area, even if the minimum heap size setting is lower. This allocation allows the artificial heap size limit imposed by the minimum heap size setting to move toward the actual heap size limit with heap expansion. See Chapter 2, "Memory management," on page 7 for more information.

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

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

The AIX 32-bit JVM default memory models

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

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

Monitoring the native heap

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

Use svmon like this:

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

Use the -r flag to print the address range.

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

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

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

  VsId      Esid Type Description          LPage  Inuse   Pin Pgspace Virtual
  50e9      -  work                -    41382    0   0  41382
                        Addr Range: 0..41381
```

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

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

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

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

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

Native heap usage

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

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

You can monitor the JIT compiling of code to avoid confusing this behavior with a memory leak. To do this, run with the command-line option

-Xjit:verbose={compileStart|compileEnd}. This command causes each method name to print to stderr as it is being compiled and, as it finishes compiling, the location in memory where the compiled code is stored.

```
(warm) Compiling java/lang/System.getEncoding(I)Ljava/lang/String;
+ (warm) java/lang/System.getEncoding(I)Ljava/lang/String; @ 0x02BA0028-0x02BA0113
      (2) Compiling java/lang/String.hashCode()I
+ (warm) java/lang/String.hashCode()I @ 0x02BA0150-0x02BA0229
      (2) Compiling java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object;)
      Ljava/lang/Object;
+ (warm) java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object;)
      Ljava/lang/Object; @ 0x02BA0270-0x02BA03F7
      (2) Compiling java/lang/String.charAt(I)C
+ (warm) java/lang/String.charAt(I)C @ 0x02BA0430-0x02BA04AC
```

```
(2) Compiling java/util/Locale.toLowerCase(Ljava/lang/String;)
    Ljava/lang/String;
+ (warm) java/util/Locale.toLowerCase(Ljava/lang/String;)Ljava/lang/String;
    @ 0x02BA04D0-0x02BA064C
```

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

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

Specifying MALLOCTYPE

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

For more information, see: System Memory Allocation Using the `malloc` Subsystem.

Monitoring the Java heap

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

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

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

where:

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

See Chapter 28, “Garbage Collector diagnostics,” on page 329 for more information about verbose GC output and monitoring.

Receiving OutOfMemoryError exceptions

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

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

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

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

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

Is the Java or native heap exhausted?

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

If a received OutOfMemory condition has one of these codes or messages, consulting Appendix C, “Messages,” on page 419 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 Chapter 2, “Memory management,” on page 7 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 Chapter 2, “Memory management,” on page 7 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 Chapter 23, “Using Heapdump,” on page 257. Similar information can be gained by using other tools, such as JProbe and OptimizIt.

Native heap exhaustion

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

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

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

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

AIX fragmentation problems

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

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

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

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

MALLOCTYPE=3.1

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

MALLOCMULTIHEAP=heaps:n,considersize

By default, the malloc subsystem uses a single heap. **MALLOCMULTIHEAP** lets users enable the use of multiple heaps of memory. Multiple heaps of memory can lead to memory fragmentation, and so the use of this environment variable is to be avoided.

MALLOCTYPE=buckets

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

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

MALLOCBUCKETS=

number_of_buckets:128,bucket_sizing_factor:64,blocks_per_bucket:1024:
bucket_statistics: <path name of file for malloc statistics>

See **MALLOCTYPE=buckets**.

Submitting a bug report

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

The process for raising a bug is detailed in Part 2, “Submitting problem reports,” on page 81, and the data that should be included in the bug report is listed as follows:

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

Debugging performance problems

Locating the causes of poor performance is often difficult. Although many factors can affect performance, the overall effect is generally perceived as poor response or slow execution of your program.

Correcting one performance problem might cause more problems in another area. By finding and correcting a bottleneck in one place you might only shift the cause of poor performance to other areas. To improve performance, experiment with tuning different parameters, monitoring the effect, and retuning until you are satisfied that your system is performing acceptably

Finding the bottleneck

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

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

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

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

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

```
> vmstat 1 10
```

outputs:

kthr		memory			page				faults				cpu			
r	b	avm	fre	re	pi	po	fr	sr	cy	in	sy	cs	us	sy	id	wa
0	0	189898	612	0	0	0	3	11	0	178	606	424	6	1	92	1
1	0	189898	611	0	1	0	0	0	0	114	4573	122	96	4	0	0
1	0	189898	611	0	0	0	0	0	0	115	420	102	99	0	0	0
1	0	189898	611	0	0	0	0	0	0	115	425	91	99	0	0	0
1	0	189898	611	0	0	0	0	0	0	114	428	90	99	0	0	0
1	0	189898	610	0	1	0	0	0	0	117	333	102	97	3	0	0
1	0	189898	610	0	0	0	0	0	0	114	433	91	99	1	0	0
1	0	189898	610	0	0	0	0	0	0	114	429	94	99	1	0	0
1	0	189898	610	0	0	0	0	0	0	115	437	94	99	0	0	0
1	0	189898	609	0	1	0	0	0	0	116	340	99	98	2	0	0

The example above shows a system that is CPU bound. This can be seen as the user (us) plus system (sy) CPU values either equal or are approaching 100. A system that is memory bound shows significant values of page in (pi) and page out (po). A system that is disk I/O bound will show an I/O wait percentage (wa) exceeding 10%. More details of vmstat can be found in “AIX debugging commands” on page 90.

CPU bottlenecks

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

The recommended tool is tprof:

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

outputs:

Mon Nov 28 12:40:11 2005

System: AIX 5.2 Node: voodoo Machine: 00455F1B4C00

Starting Command sleep 60

stopping trace collection

Generating sleep.prof

```
> cat sleep.prof
```

Process	Freq	Total	Kernel	User	Shared	Other
=====	=====	=====	=====	=====	=====	=====
./java	5	59.39	24.28	0.00	35.11	0.00
wait	4	40.33	40.33	0.00	0.00	0.00
/usr/bin/tprof	1	0.20	0.02	0.00	0.18	0.00
/etc/syncd	3	0.05	0.05	0.00	0.00	0.00
/usr/bin/sh	2	0.01	0.00	0.00	0.00	0.00
gil	2	0.01	0.01	0.00	0.00	0.00
afsd	1	0.00	0.00	0.00	0.00	0.00
rpc.lockd	1	0.00	0.00	0.00	0.00	0.00
swapper	1	0.00	0.00	0.00	0.00	0.00
=====	=====	=====	=====	=====	=====	=====
Total	20	100.00	64.70	0.00	35.29	0.00

Process	PID	TID	Total	Kernel	User	Shared	Other
=====	=====	=====	=====	=====	=====	=====	=====
./java	467018	819317	16.68	5.55	0.00	11.13	0.00
./java	467018	766019	14.30	6.30	0.00	8.00	0.00
./java	467018	725211	14.28	6.24	0.00	8.04	0.00
./java	467018	712827	14.11	6.16	0.00	7.94	0.00
wait	20490	20491	10.24	10.24	0.00	0.00	0.00
wait	8196	8197	10.19	10.19	0.00	0.00	0.00
wait	12294	12295	9.98	9.98	0.00	0.00	0.00
wait	16392	16393	9.92	9.92	0.00	0.00	0.00
/usr/bin/tprof	421984	917717	0.20	0.02	0.00	0.18	0.00
/etc/syncd	118882	204949	0.04	0.04	0.00	0.00	0.00
./java	467018	843785	0.03	0.02	0.00	0.00	0.00

gil	53274	73765	0.00	0.00	0.00	0.00	0.00
gil	53274	61471	0.00	0.00	0.00	0.00	0.00
/usr/bin/sh	397320	839883	0.00	0.00	0.00	0.00	0.00
rpc.lockd	249982	434389	0.00	0.00	0.00	0.00	0.00
/usr/bin/sh	397318	839881	0.00	0.00	0.00	0.00	0.00
swapper	0	3	0.00	0.00	0.00	0.00	0.00
afsd	65776	274495	0.00	0.00	0.00	0.00	0.00
/etc/syncd	118882	258175	0.00	0.00	0.00	0.00	0.00
/etc/syncd	118882	196839	0.00	0.00	0.00	0.00	0.00
=====	===	===	=====	=====	=====	=====	=====
Total			100.00	64.70	0.00	35.29	0.00

Total Samples = 24749 Total Elapsed Time = 61.88s

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

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

Total

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

Kernel

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

User

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

Shared

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

Other

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

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

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

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

Shared Object	%
=====	=====
/j9vmap3223-20051123/inst.images/rios_aix32_5/sdk/jre/bin/libj9gc23.so	17.42
/usr/lib/libc.a[shr.o]	9.38
/usr/lib/libpthreads.a[shr_xpg5.o]	6.94
j9vmap3223-20051123/inst.images/rios_aix32_5/sdk/jre/bin/libj9thr23.so	1.03
j9vmap3223-20051123/inst.images/rios_aix32_5/sdk/jre/bin/libj9prt23.so	0.24
/j9vmap3223-20051123/inst.images/rios_aix32_5/sdk/jre/bin/libj9vm23.so	0.10
j9vmap3223-20051123/inst.images/rios_aix32_5/sdk/jre/bin/libj9ute23.so	0.06

```
j9vmap3223-20051123/inst.images/rios_aix32_5/sdk/jre/bin/libj9jit23.so    0.05
/usr/lib/libtrace.a[shr.o]                                              0.04
j9vmap3223-20051123/inst.images/rios_aix32_5/sdk/jre/bin/libj9trc23.so    0.02
p3223-20051123/inst.images/rios_aix32_5/sdk/jre/bin/libj9hookable23.so    0.01
```

This section shows that almost all of the time is being spent in one particular shared library, which is part of the JVM installation: `libj9gc23.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:

libbcbv23.so

Bytecode Verifier

libdbg23.so

Debug Server (used by the Java Debug Interface)

libj9gc23.so

Garbage Collection

libj9jextract.so

The dump extractor, used by the `jextract` command

libj9jit23.so

The Just In Time (JIT) Compiler

libj9jvmti23.so

The JVMTI interface

libj9prt23.so

The “port layer” between the JVM and the Operating System

libj9shr23.so

The shared classes library

libj9thr23.so

The threading library

libj9ute23.so

The trace engine

libj9vm23.so

The core Virtual Machine

libj9zlib23.so

The compressed file utility library

libjclscar_23.so

The Java Class Library (JCL) support routines

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

Again, you can obtain a more accurate understanding of what is occurring inside the `libj9gc23.so` library during the CPU time by analyzing the relevant section of the `tprof` output:

```
Profile: /work/j9vmap3223-20051123/inst.images/rios_aix32_5/sdk/jre/bin/
libj9gc23.so
```

```
Total % For All Processes (/work/j9vmap3223-20051123/inst.images/rios_aix32_5/
sdk/jre/bin/libj9gc23.so) = 17.42
```

```
Subroutine                                %    Source
=====
```

Scheme::scanMixedObject(MM_Environment*,J9Object*)	2.67	MarkingScheme.cpp
MarkingScheme::scanClass(MM_Environment*,J9Class*)	2.54	MarkingScheme.cpp
.GC_ConstantPoolObjectSlotIterator::nextSlot()	1.96	ObjectSlotIterator.cpp
ParallelTask::handleNextWorkUnit(MM_EnvironmentModron*)	1.05	ParallelTask.cpp
WorkPackets::getPacket(MM_Environment*,MM_Packet*)	0.70	WorkPackets.cpp
MarkingScheme::fixupRegion(J9Object*,J9Object*,bool,long&)	0.67	CompactScheme.cpp
WorkPackets::putPacket(MM_Environment*,MM_Packet*)	0.47	WorkPackets.cpp
MarkingScheme::scanObject(MM_Environment*,J9Object*)	0.43	MarkingScheme.cpp
sweepChunk(MM_Environment*,MM_ParallelSweepChunk*)	0.42	ParallelSweepScheme.cpp
MarkingScheme::scanObject(MM_Environment*,J9Object*,unsigned long)	0.38	MarkingScheme.cpp
CompactScheme::getForwardingPtr(J9Object*) const	0.36	CompactScheme.cpp
ObjectSlotIteratorAddressOrderedList::nextObject()	0.33	AddressOrderedList.cpp
WorkPackets::getInputPacketFromOverflow(MM_Environment*)	0.32	WorkPackets.cpp
.MM_WorkStack::popNowait(MM_Environment*)	0.31	WorkStack.cpp
WorkPackets::getInputPacketNowait(MM_Environment*)	0.29	WorkPackets.cpp
canReferenceMixedObject(MM_Environment*,J9Object*)	0.29	MarkingScheme.cpp
MarkingScheme::markClass(MM_Environment*,J9Class*)	0.27	MarkingScheme.cpp
.ptrgl	0.26	ptrgl.s
MarkingScheme::initializeMarkMap(MM_Environment*)	0.25	MarkingScheme.cpp
.MM_HeapVirtualMemory::getHeapBase()	0.23	HeapVirtualMemory.cpp

This output shows that the most-used functions are:

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

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

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

In the case of CPU time being attributed to “Other”, you can use a Javdump 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 Javdump 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 Javdump.

For the example above:

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

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

```

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 `libj9gc23.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
15552	X	14282	1407	17266	19810	N	N
14762	dtwm	3991	1403	5054	7628	N	N
15274	dtsessi	3956	1403	5056	7613	N	N
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 ($144805 * 4 \text{ KB} = 565.64 \text{ MB}$). 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 **vm tune** command to alter the **maxperm** and **minperm** values.

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

I/O bottlenecks

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

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

JVM heap sizing

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

See “How to do heap sizing” on page 21 for information on how to correctly set the size of your heap.

JIT compilation and performance

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

The JIT is another area that can affect the performance of your program. The performance of short-running applications can be improved by using the **-Xquickstart** command-line parameter. The JIT is switched on by default, but you can use **-Xint** to turn it off. You also have considerable flexibility in controlling JIT processing. For more details about the JIT, see Chapter 5, “The JIT compiler,” on page 35 and Chapter 26, “JIT problem determination,” on page 317.

Application profiling

You can learn a lot about your Java application by using the hprof profiling agent

Statistics about CPU and memory usage are presented along with many other options. The hprof tool is discussed in detail in Chapter 32, “Using the HPROF Profiler,” on page 385. **-Xrunhprof:help** gives you a list of suboptions that you can use with hprof.

MustGather information for AIX

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

The AIX core file

If the environment is correctly set up to produce full AIX Core files (as detailed in “Setting up and checking your AIX environment” on page 87), a core file is generated when the process receives a terminal signal (that is, SIGSEGV, SIGILL, or SIGABORT). The core file is generated into the current working directory of the process, or at the location pointed to by the label field specified using **-Xdump**.

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

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

When a core file is generated:

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

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

to generate a file called dumpfilename.zip in the current directory. This file is compressed and contains the required files. Running jextract against the core file also allows the subsequent use of the Dump Viewer

2. If the jextract processing fails, use the snapcore utility to collect the same information. For example, `snapcore -d /tmp/savedir core.001 /usr/java5/jre/bin/java` creates an archive (snapcore_pid.pax.Z) in the file /tmp/savedir.

You also have the option of looking directly at the core file by using dbx. However, dbx does not have the advantage of understanding Java frames

and the JVM control blocks that the Dump Viewer does. Therefore, you are recommended to use the Dump Viewer in preference to dbx.

The javacore file:

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

The Error Report

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

Chapter 11. Linux problem determination

This section describes problem determination on Linux for the IBM SDK for Java, v5.0.

The topics are:

- “Setting up and checking your Linux environment”
- “General debugging techniques” on page 123
- “Diagnosing crashes” on page 129
- “Debugging hangs” on page 130
- “Debugging memory leaks” on page 131
- “Debugging performance problems” on page 131
- “MustGather information for Linux” on page 134
- “Known limitations on Linux” on page 136

Use the man command to obtain reference information about many of the commands mentioned in this set of topics.

Setting up and checking your Linux environment

Linux operating systems undergo a large number of patches and updates.

IBM personnel cannot test the JVM against every patch. The intention is to test against the most recent releases of a few distributions. In general, you should keep systems up-to-date with the latest patches. See <http://www.ibm.com/developerworks/java/jdk/linux/tested.html> for an up-to-date list of releases and distributions that have been successfully tested against.

The Java service team has a tool named ReportEnv that plugs into your JVM and reports on the JVM environment in real time. Your JVM environment affects the operation of the JVM. ReportEnv reports on environment variables and command-line parameters. It is a GUI tool, although it can be run without a GUI. The GUI allows you to browse your environment and, to some extent, dynamically change it. The tool also has a mechanism to generate reports to tell you the exact state of your JVM environment. A screenshot of the tool is shown in “Setting up and checking your Windows environment” on page 139. The ReportEnv tool is available on request from jvmcookbook@uk.ibm.com.

Working directory

The current working directory of the JVM process is the default location for the generation of core files, Java dumps, heap dumps, and the JVM trace outputs, including Application Trace and Method trace. Enough free disk space must be available for this directory. Also, the JVM must have write permission.

Linux system dumps (core files)

When a crash occurs, the most important diagnostic data to obtain is the system dump. To ensure that this file is generated, you must check the following settings.

Operating system settings

Operating system settings must be correct. These settings can vary by distribution and Linux version.

To obtain full core files, set the following ulimit options:

```
ulimit -c unlimited    turn on corefiles with unlimited size
ulimit -n unlimited    allows an unlimited number of open file descriptors
ulimit -m unlimited    sets the user memory limit to unlimited
ulimit -f unlimited    sets the file size to unlimited
```

The current ulimit settings can be displayed using:

```
ulimit -a
```

These values are the "soft" limit, and are set for each user. These values cannot exceed the "hard" limit value. To display and change the "hard" limits, the same ulimit commands can be run using the additional **-H** flag. From Java 5, the ulimit -c value for the soft limit is ignored and the hard limit value is used to help ensure generation of the core file. You can disable core file generation by using the **-Xdump:system:none** command-line option.

Java Virtual Machine settings

To generate core files when a crash occurs, check that the JVM is set to do so.

Run `java -Xdump:what`, which should produce the following:

```
dumpFn=doSystemDump
events=gpf+abort
filter=
label=/mysdk/sdk/jre/bin/core.%Y%m%d.%H%M%S.%pid.dmp
range=1..0
priority=999
request=serial
opts=
```

The values above are the default settings. At least `events=gpf` must be set to generate a core file when a crash occurs. You can change and set options with the command-line option

-Xdump:system[:name1=value1,name2=value2 ...]

Available disk space

The available disk space must be large enough for the core file to be written.

The JVM allows the core file to be written to any directory that is specified in the `label` option. For example:

```
-Xdump:system:label=/mysdk/sdk/jre/bin/core.%Y%m%d.%H%M%S.%pid.dmp
```

To write the core file to this location, disk space must be sufficient (up to 4 GB might be required for a 32-bit process), and the correct permissions for the Java process to write to that location.

ZipException or IOException on Linux

When using a large number of file descriptors to load different instances of classes, you might see an error message "java.util.zip.ZipException: error in opening zip file", or some other form of `IOException` advising that a file could not be opened. The solution is to increase the provision for file descriptors, using the `ulimit` command. To find the current limit for open files, use the command:

```
ulimit -a
```

To allow more open files, use the command:

```
ulimit -n 8196
```

Threading libraries

The distributions supported by the IBM JVM provide the enhanced Native POSIX Threads Library for Linux (NPTL).

For information on the threading libraries that are supported by the IBM Virtual Machine for Java on specific Linux platforms, see <http://www.ibm.com/developerworks/java/jdk/linux/tested.html>.

You can discover your glibc version by changing to the `/lib` directory and running the file `libc.so.6`. The Linux command `ldd` prints information that should help you to work out the shared library dependencies of your application.

Using CPU Time limits to control runaway tasks

Because real time threads run at high priorities and with FIFO scheduling, failing applications (typically with tight CPU-bound loops) can cause a system to become unresponsive. In a development environment it can be useful to ensure runaway tasks are killed by limiting the amount of CPU that tasks might consume. See “Linux system dumps (core files)” on page 121 for a discussion on soft and hard limit settings.

The command `ulimit -t` lists the current timeout value in CPU seconds. This value can be reduced with either `soft`, for example, `ulimit -St 900` to set the soft timeout to 15 minutes or `hard` values to stop runaway tasks.

General debugging techniques

This section provides a guide to the JVM-provided diagnostic tools and Linux commands that can be useful when you are diagnosing problems that occur with the Linux JVM.

There are several diagnostic tools available with the JVM to help diagnose problems:

- Starting Javadumps, see Chapter 22, “Using Javacore,” on page 245.
- Starting Heapdumps, see Chapter 23, “Using Heapdump,” on page 257.
- Starting system dumps, see Chapter 24, “Using system dumps and the dump viewer,” on page 263.

Linux provides various commands and tools that can be useful in diagnosing problems.

Using system dump tools

The commands `objdump` and `nm` are used to investigate and display information about system (core) dumps. If a crash occurs and a system dump is produced, these commands help you analyze the file.

About this task

Run these commands on the same workstation as the one that produced the system dumps to use the most accurate symbol information available. This output (together with the system dump, if small enough) is used by the IBM support team for Java to diagnose a problem.

objdump

Use this command to disassemble shared objects and libraries. After you have discovered which library or object has caused the problem, use `objdump` to locate the method in which the problem originates. To start `objdump`, enter: `objdump <option> <filename>`

You can see a complete list of options by typing `objdump -H`. The `-d` option disassembles contents of executable sections

nm This command lists symbol names from object files. These symbol names can be either functions, global variables, or static variables. For each symbol, the value, symbol type, and symbol name are displayed. Lower case symbol types mean the symbol is local, while upper case means the symbol is global or external. To use this tool, type: `nm <option> <system dump>`.

Examining process information

The kernel provides useful process and environment information. These commands can be used to view this information.

The ps command

On Linux, Java threads are implemented as system threads and might be visible in the process table, depending on the Linux distribution.

Running the `ps` command gives you a snapshot of the current processes. The `ps` command gets its information from the `/proc` file system. Here is an example of using `ps`:

```
ps -efwH
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
cass	1234	1231	0	Aug07	?	00:00:00	/bin/bash
cass	1555	1234	0	Aug07	?	00:00:02	java app
cass	1556	1555	0	Aug07	?	00:00:00	java app
cass	1557	1556	0	Aug07	?	00:00:00	java app
cass	1558	1556	0	Aug07	?	00:00:00	java app
cass	1559	1556	0	Aug07	?	00:00:00	java app
cass	1560	1556	0	Aug07	?	00:00:00	java app

e Specifies to select all processes.

f Ensures that a full listing is provided.

l Displays in long format.

m Shows threads if they are not shown by default.

w An output modifier that ensures a wide output.

H Useful when you are interested in Java threads because it displays a hierarchical listing. With a hierarchical display, you can determine which process is the primordial thread, which is the thread manager, and which are child threads. In the previous example, process 1555 is the primordial

thread, while process 1556 is the thread manager. All the child processes have a parent process ID pointing to the thread manager.

The top command

The top command displays the most CPU-intensive or memory-intensive processes in real time. It provides an interactive interface for manipulation of processes and allows sorting by different criteria, such as CPU usage or memory usage. Press **h** while running top to see all the available interactive commands.

The top command displays several fields of information for each process. The process field shows the total number of processes that are running, but breaks down the information into tasks that are running, sleeping, stopped, or undead. In addition to displaying PID, PPID, and UID, the top command displays information about memory usage and swap space. The mem field shows statistics on memory usage, including available memory, free memory, used memory, shared memory, and memory used for buffers. The swap field shows total swap space, available swap space, and used swap space.

The vmstat command

The vmstat command reports virtual storage statistics. It is useful to perform a general health check on your system because it reports on the system as a whole. Commands such as top can be used to gain more specific information about the process operation.

When you use it for the first time during a session, the information is reported as averages since the last reboot. Further usage produces reports that are based on a sampling period that you can specify as an option. `vmstat 3 4` displays values every 3 seconds for a count of four times. It might be useful to start vmstat before the application, have it direct its output to a file and later study the statistics as the application started and ran.

The basic output from this command is displayed in these sections:

processes

Shows how many processes are awaiting run time, blocked, or swapped out.

memory

Shows the amount of memory (in kilobytes) swapped, free, buffered, and cached. If the free memory is going down during certain stages of your applications execution, there might be a memory leak.

swap Shows the kilobytes per second of memory swapped in from and swapped out to disk. Memory is swapped out to disk if not enough RAM is available to store it all. Large values here can be a hint that not enough RAM is available (although it is normal to get swapping when the application first starts).

io Shows the number of blocks per second of memory sent to and received from block devices.

system

Displays the interrupts and the context switches per second. There is a performance penalty associated with each context switch so a high value for this section might mean that the program does not scale well.

cpu Shows a breakdown of processor time between user time, system time, and

idle time. The idle time figure shows how busy a processor is, with a low value indicating that the processor is busy. You can use this knowledge to help you understand which areas of your program are using the CPU the most.

ldd

The Linux command **ldd** prints information that should help you to work out the shared library dependency of your application.

Tracing tools

Tracing is a technique that presents details of the execution of your program. If you are able to follow the path of execution, you will gain a better insight into how your program runs and interacts with its environment.

Also, you will be able to pinpoint locations where your program starts to deviate from its expected behavior.

Three tracing tools on Linux are **strace**, **ltrace**, and **mtrace**. The command **man strace** displays a full set of available options.

strace

The **strace** tool traces system calls. You can either use it on a process that is already available, or start it with a new process. **strace** records the system calls made by a program and the signals received by a process. For each system call, the name, arguments, and return value are used. **strace** allows you to trace a program without requiring the source (no recompilation is required). If you use **strace** with the **-f** option, it will trace child processes that have been created as a result of a forked system call. You can use **strace** to investigate plug-in problems or to try to understand why programs do not start properly.

To use **strace** with a Java application, type **strace java <class-name>**.

You can direct the trace output from the **strace** tool to a file by using the **-o** option.

ltrace

The **ltrace** tool is distribution-dependent. It is very similar to **strace**. This tool intercepts and records the dynamic library calls as called by the executing process. **strace** does the same for the signals received by the executing process.

To use **ltrace** with a Java application, type **ltrace java <class-name>**

mtrace

mtrace is included in the GNU toolset. It installs special handlers for **malloc**, **realloc**, and **free**, and enables all uses of these functions to be traced and recorded to a file. This tracing decreases program efficiency and should not be enabled during normal use. To use **mtrace**, set **IBM_MALLOCTRACE** to 1, and set **MALLOC_TRACE** to point to a valid file where the tracing information will be stored. You must have write access to this file.

To use **mtrace** with a Java application, type:

```
export IBM_MALLOCTRACE=1
export MALLOC_TRACE=/tmp/file
java <class-name>
mtrace /tmp/file
```

Debugging with gdb

The GNU debugger (gdb) allows you to examine the internals of another program while the program executes or retrospectively to see what a program was doing at the moment that it crashed.

The gdb allows you to examine and control the execution of code and is useful for evaluating the causes of crashes or general incorrect behavior. gdb does not handle Java processes, so it is of limited use on a pure Java program. It is useful for debugging native libraries and the JVM itself.

Running gdb

You can run gdb in three ways:

Starting a program

Typically the command: `gdb <application>` is used to start a program under the control of gdb. However, because of the way that Java is launched, you must start gdb by setting an environment variable and then calling Java:

```
export IBM_JVM_DEBUG_PROG=gdb
java
```

Then you receive a gdb prompt, and you supply the run command and the Java arguments:

```
r <java_arguments>
```

Attaching to a running program

If a Java program is already running, you can control it under gdb. The process ID of the running program is required, and then gdb is started with the Java application as the first argument and the process ID as the second argument:

```
gdb <Java Executable> <PID>
```

When gdb is attached to a running program, this program is halted and its position in the code is displayed for the viewer. The program is then under the control of gdb and you can start to issue commands to set and view the variables and generally control the execution of the code.

Running on a system dump (corefile)

A system dump is typically produced when a program crashes. gdb can be run on this system dump. The system dump contains the state of the program when the crash occurred. Use gdb to examine the values of all the variables and registers leading up to a crash. This information helps you discover what caused the crash. To debug a system dump, start gdb with the Java application file as the first argument and the system dump name as the second argument:

```
gdb <Java Executable> <system dump>
```

When you run gdb against a system dump, it initially shows information such as the termination signal the program received, the function that was executing at the time, and even the line of code that generated the fault.

When a program comes under the control of gdb, a welcome message is displayed followed by a prompt (gdb). The program is now waiting for you to enter instructions. For each instruction, the program continues in whichever way you choose.

Setting breakpoints and watchpoints

Breakpoints can be set for a particular line or function using the command:

```
break lineNumber
```

or

```
break functionName
```

After you have set a breakpoint, use the `continue` command to allow the program to execute until it reaches a breakpoint.

Set breakpoints using conditionals so that the program halts only when the specified condition is reached. For example, using breakpoint 39 `if var == value` causes the program to halt when it reaches line 39, but only if the variable is equal to the specified value.

If you want to know *where* as well as *when* a variable became a certain value you can use a watchpoint. Set the watchpoint when the variable in question is in scope. After doing so, you will be alerted whenever this variable attains the specified value. The syntax of the command is: `watch var == value`.

To see which breakpoints and watchpoints are set, use the `info` command:

```
info break  
info watch
```

When `gdb` reaches a breakpoint or watchpoint, it prints out the line of code it is next set to execute. Setting a breakpoint at line 8 will cause the program to halt after completing execution of line 7 but before execution of line 8. As well as breakpoints and watchpoints, the program also halts when it receives certain system signals. By using the following commands, you can stop the debugging tool halting every time it receives these system signals:

```
handle sig32 pass nostop noprint  
handle sigusr2 pass nostop noprint
```

Examining the code

When the correct position of the code has been reached, there are a number of ways to examine the code. The most useful is `backtrace` (abbreviated to `bt`), which shows the call stack. The call stack is the collection of function frames, where each function frame contains information such as function parameters and local variables. These function frames are placed on the call stack in the order that they are executed. This means that the most recently called function is displayed at the top of the call stack. You can follow the trail of execution of a program by examining the call stack. When the call stack is displayed, it shows a frame number on the left side, followed by the address of the calling function, followed by the function name and the source file for the function. For example:

```
#6 0x804c4d8 in myFunction () at myApplication.c
```

To view more detailed information about a function frame, use the `frame` command along with a parameter specifying the frame number. After you have selected a frame, you can display its variables using the command `print var`.

Use the `print` command to change the value of a variable; for example, `print var = newValue`.

The `info locals` command displays the values of all local variables in the selected function.

To follow the exact sequence of execution of your program, use the `step` and `next` commands. Both commands take an optional parameter specifying the number of lines to execute. However, `next` treats function calls as a single line of execution, while `step` progresses through each line of the called function, one step at a time.

Useful commands

When you have finished debugging your code, the `run` command causes the program to run through to its end or its crash point. The `quit` command is used to exit `gdb`.

Other useful commands are:

ptype

Prints data type of variable.

info share

Prints the names of the shared libraries that are currently loaded.

info functions

Prints all the function prototypes.

list

Shows the 10 lines of source code around the current line.

help

Displays a list of subjects, each of which can have the `help` command called on it, to display detailed help on that topic.

Diagnosing crashes

Many approaches are possible when you are trying to determine the cause of a crash. The process typically involves isolating the problem by checking the system setup and trying various diagnostic options.

Checking the system environment

The system might have been in a state that has caused the JVM to crash. For example, this could be a resource shortage (such as memory or disk) or a stability problem. Check the Jvadbump file, which contains various system information (as described in Chapter 22, “Using Jvadbump,” on page 245). The Jvadbump file tells you how to find disk and memory resource information. The system logs can give indications of system problems.

Gathering process information

It is useful to find out what exactly was happening leading up to the crash.

Analyze the core file (as described in Chapter 24, “Using system dumps and the dump viewer,” on page 263) to produce a stack trace, which will show what was running up to the point of the crash. This could be:

- JNI native code.
- JIT or AOT compiled code. If you have a problem with JIT or AOT code, try running without the JIT or AOT code by using the `-Xint` option.
- JVM code.

Other tracing methods:

- ltrace
- strace
- mtrace - can be used to track memory calls and determine possible corruption
- RAS trace, described in Chapter 31, “Using the Reliability, Availability, and Serviceability Interface,” on page 371.

Finding out about the Java environment

Use the Jvareadump to determine what each thread was doing and which Java methods were being executed. Match function addresses against library addresses to determine the source of code executing at various points.

Use the **-verbose:gc** option to look at the state of the Java heap and determine if:

- There was a shortage of Java heap space and if this could have caused the crash.
- The crash occurred during garbage collection, indicating a possible garbage collection fault. See Chapter 28, “Garbage Collector diagnostics,” on page 329.
- The crash occurred after garbage collection, indicating a possible memory corruption.

For more information about memory management, see Chapter 2, “Memory management,” on page 7.

Debugging hangs

A hang is caused by a wait (also known as a deadlock) or a loop (also known as a livelock). A deadlock sometimes occurs because of a wait on a lock or monitor. A loop can occur similarly or sometimes because of an algorithm making little or no progress towards completion.

A wait could either be caused by a timing error leading to a missed notification, or by two threads deadlocking on resources.

For an explanation of deadlocks and diagnosing them using a Jvareadump, see “Locks, monitors, and deadlocks (LOCKS)” on page 252.

A loop is caused by a thread failing to exit a loop in a timely manner. The problem might occur because the thread calculated the wrong limit value, or missed a flag that was intended to exit the loop. If the problem occurs only on multiprocessor workstations, the failure can usually be traced to:

- A failure to make the flag volatile.
- A failure to access the flag while holding an appropriate monitor.

The following approaches are useful to resolve waits and loops:

- Monitoring process and system state (as described in “MustGather information for Linux” on page 134).
- Jvareadumps give monitor and lock information. You can trigger a Jvareadump during a hang by using the `kill -QUIT <PID>` command.
- **-verbose:gc** information is useful. It indicates:
 - Excessive garbage collection, caused by a lack of Java heap space, which makes the system seem to be in livelock

- Garbage collection causing a hang or memory corruption which later causes hangs

Debugging memory leaks

If dynamically allocated objects are not freed at the end of their lifetime, memory leaks can occur. When objects that should have had their memory released are still holding memory and more objects are being created, the system eventually runs out of memory.

The **mtrace** tool from GNU is available for tracking memory calls. This tool enables you to trace memory calls such as `malloc` and `realloc` so that you can detect and locate memory leaks.

For more details about analyzing the Java Heap, see Chapter 23, “Using Heapdump,” on page 257.

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

Given that any performance problem could be caused by any one of several factors, you must look at several areas to eliminate each one.

Determine which resource is constraining the system:

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

Several tools are available that enable you to measure system components and establish how they are performing and under what kind of workload.

The key things to measure are CPU usage and memory usage. If the CPU is not powerful enough to handle the workload, it will be impossible to tune the system to make much difference to overall performance. You must upgrade the CPU. Similarly, if a program is running in an environment without enough memory, an increase in the memory improves performance far more than any amount of tuning.

CPU usage

Java processes consume 100% of processor time when they reach their resource limits. Ensure that `ulimit` settings are appropriate to the application requirement.

See “Linux system dumps (core files)” on page 121 for more information about `ulimit`.

The `/proc` file system provides information about all the processes that are running on your system, including the Linux kernel. See `man proc` from a Linux shell for official Linux documentation about the `/proc` file system.

The `top` command provides real-time information about your system processes. The `top` command is useful for getting an overview of the system load. It clearly displays which processes are using the most resources. Having identified the processes that are probably causing a degraded performance, you can take further steps to improve the overall efficiency of your program. More information is provided about the `top` command in “The `top` command” on page 125.

Memory usage

If a system is performing poorly because of lack of memory resources, it is memory bound. By viewing the contents of `/proc/meminfo`, you can view your memory resources and see how they are being used. `/proc/swap` contains information on your swap file.

Swap space is used as an extension of the systems virtual storage. Therefore, not having enough memory or swap space causes performance problems. A general guideline is that swap space should be at least twice as large as the physical memory.

A swap space can be either a file or disk partition. A disk partition offers better performance than a file does. `fdisk` and `cfdisk` are the commands that you use to create another swap partition. It is a good idea to create swap partitions on different disk drives because this distributes the I/O activities and thus reduces the chance of further bottlenecks.

The `vmstat` tool helps you find where performance problems might be caused. For example, if you see that high swap rates are occurring, you probably do not have enough physical or swap space. The `free` command displays your memory configuration; `swapon -s` displays your swap device configuration. A high swap rate (for example, many page faults) means that you probably need to increase your physical memory. More information about the `vmstat` command are provided in “The `vmstat` command” on page 125.

Network problems

Another area that often affects performance is the network. The more you know about the behavior of your program, the easier it is to decide whether the network might be a performance bottleneck.

If you think that your program is likely to be network I/O bound, `netstat` is a useful tool. The `netstat` command provides information about network routes, active sockets for each network protocol, and statistics, such as the number of packets that are received and sent.

Use `netstat` to see how many sockets are in a `CLOSE_WAIT` or `ESTABLISHED` state. You can tune the TCP/IP parameters accordingly for better performance of the system. For example, tuning `/proc/sys/net/ipv4/tcp_keepalive_time` reduces the time for socket waits in `TIMED_WAIT` state before closing a socket.

If you are tuning the `/proc/sys/net` file system, the changes affect all the applications running on the system. To change an individual socket or connection, use Java Socket API calls on the appropriate socket object. Use `netstat -p`, or the

`lsof` command, to find the PID of the process that owns a particular socket. Use the `kill -QUIT <pid>` command to generate a `javacore` file that contains details of the socket object in the stack trace.

You can also use the option **-Xtrace:print=net**, to trace out network-related activity in the JVM. This technique is helpful when socket-related Java thread hangs are seen. Correlating output from `netstat -p`, `lsof`, JVM net trace, and `ps -efH` can help you to diagnose the network-related problems.

Providing summary statistics that are related to your network is useful for investigating programs that might be under-performing because of TCP/IP problems. The more you understand your hardware capacity, the easier it is to tune the parameters of system components that improve the performance of your application. You can also determine whether tuning the system improves performance or whether you require system upgrades.

JVM heap sizing

The Java heap size is one of the most important tuning parameters of your JVM. Choose the correct size to optimize performance. Using the correct size can make it easier for the Garbage Collector to provide the required utilization.

See “How to do heap sizing” on page 21 for information on how to correctly set the size of your heap.

JIT compilation and performance

The JIT is another area that can affect the performance of your program. When deciding whether or not to use JIT compilation, you must make a balance between faster execution and increased processor usage during compilation.

The performance of short-running applications can be improved by using the **-Xquickstart** command-line parameter. The JIT is switched on by default, but you can use **-Xint** to turn it off. You also have considerable flexibility in controlling JIT processing. For more details about the JIT, see Chapter 5, “The JIT compiler,” on page 35 and Chapter 26, “JIT problem determination,” on page 317.

Application profiling

You can learn a lot about your Java application by using the `hprof` profiling agent. Statistics about CPU and memory usage are presented along with many other options.

For information about the `hprof` tool, see Chapter 32, “Using the HPROF Profiler,” on page 385.

The **-Xrunhprof:help** command-line option shows you a list of suboptions that you can use with `hprof`.

The Performance Inspector package contains a suite of performance analysis tools for Linux. You can use tools to help identify performance problems in your application and understand how your application interacts with the Linux kernel. See <http://perfinsp.sourceforge.net/> for details.

MustGather information for Linux

When a problem occurs, the more information known about the state of the system environment, the easier it is to reach a diagnosis of the problem.

A large set of information can be collected, although only some of it will be relevant for particular problems. The following sections tell you the data to collect to help the IBM service team for Java solve the problem.

Collecting system dumps (core files)

Collect system dumps to help diagnose many types of problem. Process the system dump with `jextract`. The resultant xml file is useful for service (see “Using the dump extractor, `jextract`” on page 264).

Producing system dumps

You can use the `-Xdump:system` command line option to obtain system dumps based on a trigger. See Chapter 21, “Using dump agents,” on page 223 for more information.

You can also use a Linux system utility to generate system dumps:

1. Determine the Process ID of your application using the `ps` command. See “The `ps` command” on page 124.
2. At a shell prompt, type `gcore -o <dump file name> <pid>`

A system dump file is produced for your application. The application will be suspended while the system dump is written.

Process the system dump with `jextract`. The resultant jar file is useful for service (see “Using the dump extractor, `jextract`” on page 264).

Producing Javadumps

In some conditions, a crash, for example, a Jav_dump is produced, usually in the current directory.

In others for example, a hang, you might have to prompt the JVM for this by sending the JVM a SIGQUIT symbol:

1. Determine the Process ID of your application using the `ps` command. See “The `ps` command” on page 124.
2. At a shell prompt, type `kill -QUIT <pid>`

This is discussed in more detail in Chapter 22, “Using Jav_dump,” on page 245.

Producing Heapdumps

The JVM can generate a Heapdump at the request of the user, for example by calling `com.ibm.jvm.Dump.HeapDump()` from inside the application, or by default when the JVM terminates because of an `OutOfMemoryError`. You can specify finer control of the timing of a Heapdump with the `-Xdump:heap` option. For example, you could request a heapdump after a certain number of full garbage collections have occurred. The default heapdump format (phd files) is not human-readable and you process it using available tools such as `Heaproots`.

Producing Snap traces

Under default conditions, a running JVM collects a small amount of trace data in a special wraparound buffer. This data is dumped to file when the JVM terminates unexpectedly or an `OutOfMemoryError` occurs. You can use the `-Xdump:snap` option to vary the events that cause a snap trace to be produced. The snap trace is in normal trace file format and requires the use of the supplied standard trace formatter so that you can read it. See “Snap traces” on page 232 for more information about the contents and control of snap traces.

Using system logs

The kernel logs system messages and warnings. The system log is located in the `/var/log/messages` file. Use it to observe the actions that led to a particular problem or event. The system log can also help you determine the state of a system. Other system logs are in the `/var/log` directory.

Determining the operating environment

This section looks at the commands that can be useful to determine the operating environment of a process at various stages of its life-cycle.

uname -a

Displays operating system and hardware information.

df Displays free disk space on a system.

free

Displays memory use information.

ps -ef

Displays a full process list.

lsof

Displays open file handles.

top

Displays process information (such as processor, memory, states) sorted by default by processor usage.

vmstat

Displays general memory and paging information.

The `uname`, `df`, and `free` output is the most useful. The other commands can be run before and after a crash or during a hang to determine the state of a process and to provide useful diagnostic information.

Sending information to Java Support

When you have collected the output of the commands listed in the previous section, put that output into files.

Compress the files (which could be very large) before sending them to Java Support. You should compress the files at a very high ratio.

The following command builds an archive from files `{file1,...,fileN}` and compresses them to a file with a name in the format `filename.tgz`:

```
tar czf filename.tgz file1 file2...filen
```


Collecting additional diagnostic data

Depending on the type of problem, the following data can also help you diagnose problems. The information available depends on the way in which Java is started and also the system environment. You will probably have to change the setup and then restart Java to reproduce the problem with these debugging aids switched on.

/proc file system

The /proc file system gives direct access to kernel level information. The /proc/<pid> directory contains detailed diagnostic information about the process with PID (process id) <pid>, where <pid> is the id of the process.

The command `cat /proc/<pid>/maps` lists memory segments (including native heap) for a given process.

strace, ltrace, and mtrace

Use the commands `strace`, `ltrace`, and `mtrace` to collect further diagnostic data. See “Tracing tools” on page 126.

Known limitations on Linux

Linux has been under rapid development and there have been various issues with the interaction of the JVM and the operating system, particularly in the area of threads.

Note the following limitations that might be affecting your Linux system.

Threads as processes

The JVM for Linux implements Java threads as native threads. On NPTL-enabled systems such as RHEL3 and SLES9, these are implemented as threads. However using the `LinuxThreads` library results in each thread being a separate Linux process.

If the number of Java threads exceeds the maximum number of processes allowed, your program might:

- Get an error message
- Get a **SIGSEGV** error
- Stop

The native stack size is the main limitation when running many threads. Use the `-Xss` option to reduce the size of the thread stack so that the JVM can handle the required number of threads. For example, set the stack size to 32 KB on startup.

For more information, see *The Volano Report* at <http://www.volano.com/report/index.html>.

Floating stacks limitations

If you are running without floating stacks, regardless of what is set for `-Xss`, a minimum native stack size of 256 KB for each thread is provided.

On a floating stack Linux system, the `-Xss` values are used. If you are migrating from a non-floating stack Linux system, ensure that any `-Xss` values are large enough and are not relying on a minimum of 256 KB. (See also “Threading libraries” on page 123.)

glibc limitations

If you receive a message indicating that the `libjava.so` library could not be loaded because of a symbol not found (such as `__bzero`), you might have an earlier version of the GNU C Runtime Library, `glibc`, installed. The SDK for Linux thread implementation requires `glibc` version 2.3.2 or greater.

Font limitations

When you are installing on a Red Hat system, to allow the font server to find the Java TrueType fonts, run (on Linux IA32, for example):

```
/usr/sbin/chkfontpath --add /opt/ibm/java2-i386-50/jre/lib/fonts
```

You must do this at installation time and you must be logged on as “root” to run the command. For more detailed font issues, see the *Linux SDK and Runtime Environment User Guide*.

Desktop support on PPC SLES9 distributions

On 32-bit PPC SLES9 distributions, `Desktop.isDesktopSupported()` returns `true`, but `Desktop.getDesktop().open()` fails to start the file with the associated application. This problem is currently being addressed through Bugzilla defect 39754.

On 64-bit PPC SLES9 distributions, `Desktop.isDesktopSupported()` returns `false`. This is because the 64-bit version of the `libgnomevfs` library is not included on these distributions. This problem is currently being addressed through Bugzilla defect 39752.

Linux Completely Fair Scheduler affects Java performance

Java applications that use synchronization extensively might perform poorly on Linux distributions that include the Completely Fair Scheduler. The Completely Fair Scheduler (CFS) is a scheduler that was adopted into the mainline Linux kernel as of release 2.6.23. The CFS algorithm is different from the scheduling algorithms for previous Linux releases. It might change the performance properties of some applications. In particular, CFS implements `sched_yield()` differently, making it more likely that a yielding thread is given CPU time regardless.

If you encounter this problem, you might observe high CPU usage by your Java application, and slow progress through synchronized blocks. The application might seem to stop because of the slow progress.

There are two possible workarounds:

- Start the JVM with the additional argument `-Xthr:minimizeUserCPU`.
- Configure the Linux kernel to use an implementation of `sched_yield()` that is more compatible with earlier versions. Do this by setting the `sched_compat_yield` tunable kernel property to 1. For example:

```
echo "1" > /proc/sys/kernel/sched_compat_yield
```

Do not use these workarounds unless you are experiencing poor performance.

This problem might affect IBM Developer Kit and Runtime Environment for Linux 5.0 (all versions) and 6.0 (all versions up to and including SR 4) running on Linux kernels that include the Completely Fair Scheduler. For IBM Developer Kit and Runtime Environment for Linux version 6.0 after SR 4, the use of CFS in the kernel is detected and the option **-Xthr:minimizeUserCPU** enabled automatically. Some Linux distributions that include the Completely Fair Scheduler are Ubuntu 8.04 and SUSE Linux Enterprise Server 11.

More information about CFS can be found at [Multiprocessing with the Completely Fair Scheduler](#).

Chapter 12. Windows problem determination

This section describes problem determination on Windows.

The topics are:

- “Setting up and checking your Windows environment”
- “General debugging techniques” on page 141
- “Diagnosing crashes in Windows” on page 142
- “Debugging hangs” on page 143
- “Debugging memory leaks” on page 143
- “Debugging performance problems” on page 145
- “MustGather information for Windows” on page 146

Setting up and checking your Windows environment

The operation of the JRE on Windows is controlled by a number of environment variables.

If you experience initial problems in running the JVM, check the following:

PATH

The **PATH** environment variable must point to the directory of your Java installation that contains the file `java.exe`. Ensure that **PATH** includes the `\bin` directory of your Java installation.

CLASSPATH

The JRE uses this environment variable to find the classes it needs when it runs. This is useful when the class you want to run uses classes that are located in other directories. By default, this is blank. If you install a product that uses the JRE, **CLASSPATH** is automatically set to point to the jar files that the product needs.

The Java service team has a tool named ReportEnv that plugs into your JRE and reports on it. ReportEnv reports on environment variables and command-line parameters. It is a GUI tool, although it can be run without a GUI. The GUI allows you to browse your environment and, to some extent, dynamically change it. The tool also has a mechanism to generate reports to tell you the exact state of your JRE. The ReportEnv tool is available on request from jvmcookbook@uk.ibm.com.

The following screenshot shows the ReportEnv tool.

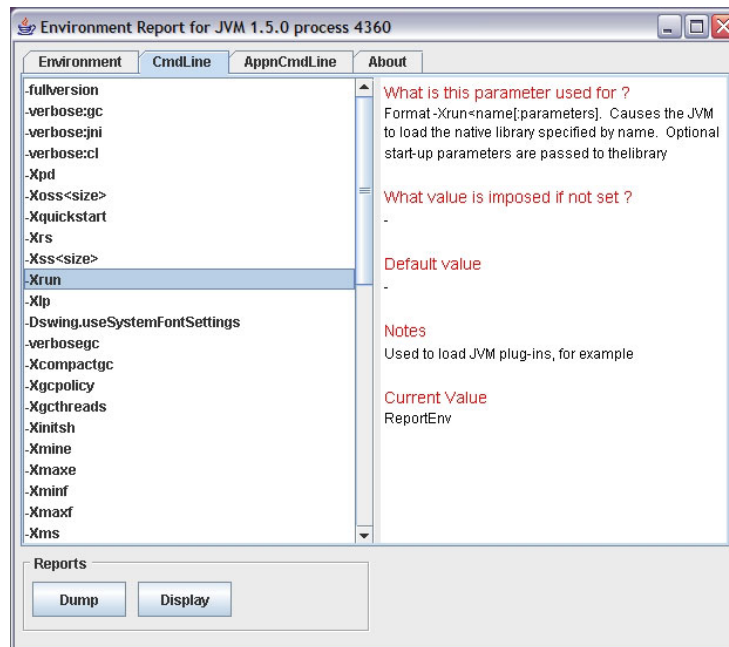


Figure 1. Screenshot of ReportEnv tool

Windows 32-bit large address aware support

The 32-bit IBM JVM for Windows includes support for the `/LARGEADDRESSAWARE` switch, also known as the `/3GB` switch. This switch increases the amount of space available to a process, from 2 GB to 3 GB. The switch is a Windows boot parameter, not a command line-option to the JVM.

This switch is useful in the following situations:

- Your application requires a very large number of threads.
- Your application requires a large amount of native memory.
- Your application has a very large codebase, causing large amounts of JIT compiled code.

To enable large address support, modify your boot.ini file and reboot your computer. See the related links for more detailed information.

After enabling the `/3GB` switch, the JVM gains 1 GB of extra memory space. This extra space does not increase the theoretical maximum size of the Java heap, but does allow the Java heap to grow closer to its theoretical maximum size (2 GB - 1 bytes), because the extra memory can be used for the native heap.

Related links

- *How to Set the /3GB Startup Switch in Windows:* <http://technet.microsoft.com/en-us/library/bb124810.aspx>.
- *Memory Support and Windows Operating Systems:* <http://www.microsoft.com/whdc/system/platform/server/PAE/PAEmem.mspx>.
- *A description of the 4 GB RAM Tuning feature and the Physical Address Extension switch:* <http://support.microsoft.com/kb/291988/>.

General debugging techniques

This section provides a guide to the JVM-provided diagnostic tools that can be useful when you are diagnosing problems that occur with the Windows JVM.

There are several diagnostic tools available with the JVM to help diagnose problems:

- Starting Javadumps, see Chapter 22, “Using Javacore,” on page 245.
- Starting Heapdumps, see Chapter 23, “Using Heapdump,” on page 257.
- Starting system dumps, see Chapter 24, “Using system dumps and the dump viewer,” on page 263.

System dump

When a JVM crash occurs, the JVM requests the operating system to generate a system dump.

A system dump consists of all the memory that is being used by the JVM; this includes the application heap, along with all JVM and user libraries. System dumps allow the IBM service personnel to look at the state of the JVM at the time of crash, and help them with the problem determination process. Because a system dump contains all of the memory allocated by the JVM process, system dump files can be very large.

You can find the location of the generated system dump in the output that is displayed in the console after the crash. Here is an example of the output:

```
Unhandled exception
Type=GPF vmState=0x00000003
Target=2_20_20040813_1848_lHdSMR (Windows 2000 5.0 build 2195 Service Pack 4)
CPU=x86 (1 logical CPUs) (0x1ff7c000 RAM)
ExceptionCode=c0000005 ExceptionAddress=1130B074 ContextFlags=0001003f
Handler1=1130B07C
Handler2=1130B080
EDI=00074af0 ESI=0000001e EAX=0006f978 EBX=00000000
ECX=00000000 EDX=00230608 EBP=0006f924
EIP=7800f4a2 ESP=0006f6cc
Module=C:\WINNT\system32\MSVCRT.dll
Module_base_address=78000000 Offset_in_DLL=0000f4a2
(I)DUMP0006 Processing Dump Event "gpf", detail "" - Please Wait.
(I)DUMP0007 JVM Requesting System Dump using 'D:\core.20040817.131302.2168.dmp'
(I)DUMP0010 System Dump written to D:\core.20040817.131302.2168.dmp
(I)DUMP0007 JVM Requesting Java Dump using 'D:\javacore.20040817.131319.2168.txt'
(I)DUMP0010 Java Dump written to D:\javacore.20040817.131319.2168.txt
(I)DUMP0013 Processed Dump Event "gpf", detail "".
```

In this example, the JVM has generated the dump in the file D:\core.20040817.131302.2168.dmp.

The JVM attempts to generate the system dump file in one of the following directories (listed in order of precedence):

1. The directory pointed to by environment variable **IBM_COREDIR**.
2. The current directory.
3. The directory pointed to by the environment variable **TMPDIR**.
4. The C:\Temp directory

Use **-Xdump:what** to find the current naming convention of all dump files. Use **-Xdump:help** to learn how to change these settings.

You might want to keep system dumps more private by setting the environment variable **IBM_COREDIR**, if you are concerned about passwords and other security details that are contained in a system dump.

Diagnosing crashes in Windows

You generally see a crash either as an unrecoverable exception thrown by Java or as a pop-up window notifying you of a General Protection Fault (GPF). The pop-up window usually refers to java.exe as the application that caused the crash. Crashes can occur because of a fault in the JRE, or because of a fault in native (JNI) code being run in the Java process.

Try to determine whether the application has any JNI code or uses any third-party packages that use JNI code (for example, JDBC application drivers, and HPROF Profiling plug-ins). If this is not the case, the fault must be in the JRE.

Try to re-create the crash with minimal dependencies (in terms of JVM options, JNI applications, or profiling tools).

In a crash condition, gather as much data as possible for the IBM service team for Java. You should:

- Collect the Javadump. See Chapter 22, “Using Javadump,” on page 245 for details.
- Collect the core dump. See “Setting up and checking your Windows environment” on page 139 for details.
- Collect the snap trace file. See Chapter 25, “Tracing Java applications and the JVM,” on page 283 for details.
- Run with the JIT turned off. See Chapter 26, “JIT problem determination,” on page 317 for details. If the problem disappears with the JIT turned off, try some JIT compile options to see if the problem can be narrowed down further.
- Try adjusting the garbage collection parameters. See Chapter 2, “Memory management,” on page 7 for details. Make a note of any changes in behavior.
- If your problem is occurring on a multiprocessor system, test your application on a uniprocessor system. You can use the BIOS options on your SMP box to reset the processor affinity to 1 to make it behave like a uniprocessor. If the problem disappears, make a note in your bug report. Otherwise, collect the crash dump.

Data to send to IBM

At this point, you potentially have several sets of either logs or dumps, or both (for example, one set for normal running, one set with JIT off, and so on).

Label the files appropriately and make them available to IBM. (See Part 2, “Submitting problem reports,” on page 81 for details.) The required files are:

- The JVM-produced Javadump file (Jvacore)
- The *dumpfile.jar* file generated by jextract

Debugging hangs

Hangs refer to the JVM locking up or refusing to respond.

A hang can occur when:

- Your application entered an infinite loop.
- A deadlock has occurred

To determine which of these situations applies, open the **Windows Task Manager** and select the **Performance** tab. If the CPU time is 100% divided by the number of processors and your system is running very slowly, the JVM is very likely to have entered an infinite loop. Otherwise, if CPU usage is normal, you are more likely to have a deadlock situation.

Getting a dump from a hung JVM

On Windows, the JVM produces a Java dump in response to a SIGBREAK signal. You can send this signal using the Ctrl-Break key combination.

You can also configure the JVM to produce a system dump on SIGBREAK by using the `-Xdump:system:events=user` option. See Chapter 21, “Using dump agents,” on page 223 for details.

If the JVM is not responding to the SIGBREAK signal, you can use the User Mode Process Dumper utility, which is available as a download from www.microsoft.com. Documentation is provided with the utility. Basic usage is as follows

userdump -p

Lists all the processes and their pids.

userdump xxx

Creates a dump file of a process that has a pid of xxx. (processname.dmp file is created in the current directory where userdump.exe is run.)

You can use the dump viewer to examine the system dump produced by this utility. See Chapter 24, “Using system dumps and the dump viewer,” on page 263 for details.

Analyzing deadlocks

A deadlocked process does not use any CPU time.

For an explanation of deadlocks and how to diagnose them using the information in the Jvaidump tool, see “Locks, monitors, and deadlocks (LOCKS)” on page 252.

Debugging memory leaks

This section begins with a discussion of the Windows memory model and the Java heap to provide background understanding before going into the details of memory leaks.

The Windows memory model

Windows memory is virtualized. Applications do not have direct access to memory addresses, so allowing Windows to move physical memory and to swap memory in and out of a swapper file (called pagefile.sys).

Allocating memory is usually a two-stage process. Just allocating memory results in an application getting a handle. No physical memory is reserved. There are more handles than physical memory. To use memory, it must be 'committed'. At this stage, a handle references physical memory. This might not be all the memory you requested.

For example, the stack allocated to a thread is usually given a small amount of actual memory. If the stack overflows, an exception is thrown and the operating system allocates more physical memory so that the stack can grow.

Memory manipulation by Windows programmers is hidden inside libraries provided for the chosen programming environment. In the C environment, the basic memory manipulation routines are the familiar malloc and free functions. Windows APIs sit on top of these libraries and generally provide a further level of abstraction.

For a programmer, Windows provides a flat memory model, in which addresses run from 0 up to the limit allowed for an application. Applications can choose to segment their memory. In a dump, the programmer sees sets of discrete memory addresses.

Classifying leaks

You can classify memory leaks from the usage of Windows memory and the size of the Java heap.

The following scenarios are possible :

- Windows memory usage is increasing and the Java heap is static:
 - Memory leak in application native code.
 - Memory leak in JRE native code.
 - Leak with hybrid Java and native objects (an unlikely occurrence).
- Windows memory usage increases because the Java heap keeps growing:
 - Memory leak in application Java code. (See “Common causes of perceived leaks” on page 329 for more information.)
 - Memory leak in JRE Java code.

Tracing leaks

Some useful techniques for tracing leaks are built into the JVM.

The techniques are:

- The **-verbose:gc** option. See “Garbage collection triggered by System.gc()” on page 332.
- HPROF tools. See Chapter 32, “Using the HPROF Profiler,” on page 385.

-Xrunjnicchk option

You can use the **-Xrunjnicchk** 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 70.

Note that `-Xrunjnichk` is equivalent to `-Xcheck:jni`. See “Debugging the JNI” on page 78 for information on the `-Xrunjnichk` suboptions.

–memorycheck option

The `-memorycheck` option can help you identify memory leaks inside the JVM. The `-memorycheck` option traces the JVM calls to the operating system's `malloc()` and `free()` functions, and identifies any JVM mistakes in memory allocation.

See General command-line options for more information.

Using Heapdump to debug memory leaks

You can use Heapdump to analyze the Java Heap.

For details about analyzing the Heap, see Chapter 23, “Using Heapdump,” on page 257.

OutOfMemoryError creating a thread

The `java.lang.OutOfMemoryError: Failed to create a thread` message occurs when the system does not have enough resources to create a new thread.

There are two possible causes of the `java.lang.OutOfMemoryError: Failed to create a thread` message:

- There are too many threads running and the system has run out of internal resources to create new threads.
- The system has run out of native memory to use for the new thread. Threads require a native memory for internal JVM structures, a Java stack, and a native stack.

To correct the problem, either:

- Increase the amount of native memory available by lowering the size of the Java heap using the `-Xmx` option.
- Lower the number of threads in your application.

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 time 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)

Windows systems resource usage

The Windows Task Manager display gives a good general view of system resource usage. You can use this tool to determine which processes are using excessive CPU time and memory. This tool also provides a summary view of network I/O activity.

For a more detailed view of Windows performance data, use the Windows Performance Monitor tool, which is provided as part of the Windows Administrative Tools. This tool provides a comprehensive view of processor, memory, and I/O device performance metrics.

JVM heap sizing

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

See “How to do heap sizing” on page 21 for information on how to correctly set the size of your heap.

JIT compilation and performance

The JIT is another area that can affect the performance of your program. When deciding whether or not to use JIT compilation, you must make a balance between faster execution and increased processor usage during compilation.

The performance of short-running applications can be improved by using the **-Xquickstart** command-line parameter. The JIT is switched on by default, but you can use **-Xint** to turn it off. You also have considerable flexibility in controlling JIT processing. For more details about the JIT, see Chapter 5, “The JIT compiler,” on page 35 and Chapter 26, “JIT problem determination,” on page 317.

Application profiling

You can learn a lot about your Java application by using the hprof profiling agent. Statistics about CPU and memory usage are presented along with many other options.

The hprof tool is discussed in detail in Chapter 32, “Using the HPROF Profiler,” on page 385. **-Xrunhprof:help** gives you a list of suboptions that you can use with hprof.

MustGather information for Windows

The more information that you can collect about a problem, the easier it is to diagnose that problem. A large set of data can be collected, although some is relevant to particular problems.

The following list describes a typical data set that you can collect to assist IBM service to fix your problem.

- Javadumps. These can be generated automatically or manually. Automatic dumps are essential for IBM service.
- Heapdumps. If generated automatically, they are essential. They are also essential if you have a memory or performance problem.
- System dump generated by the JVM. See “System dump” on page 141. This dump is the key to most problems and you collect it by running jextract against the system dump and obtaining a compressed *dumpfile.zip*
- WebSphere Application Server logs, if you are working in a WebSphere Application Server environment.
- Other data, as determined by your particular problem.

Chapter 13. z/OS problem determination

This section describes problem determination on z/OS.

The topics are:

- “Setting up and checking your z/OS environment”
- “General debugging techniques” on page 151
- “Diagnosing crashes” on page 153
- “Debugging hangs” on page 159
- “Understanding Memory Usage” on page 161
- “Debugging performance problems” on page 163
- “MustGather information for z/OS” on page 164

Setting up and checking your z/OS environment

Set up the right environment for the z/OS JVM to run correctly.

Maintenance

The Java for z/OS Web site has up-to-date information about any changing operating system prerequisites for correct JVM operation. In addition, any new prerequisites are described in PTF HOLDDATA.

The Web site is at:

<http://www.ibm.com/servers/eserver/zseries/software/java/>

LE settings

Language Environment® (LE) Runtime Options (RTOs) affect operation of C and C++ programs such as the JVM. In general, the options provided by IBM using C `#pragma` statements in the code must not be overridden because they are generated as a result of testing to provide the best operation of the JVM.

Environment variables

Environment variables that change the operation of the JVM in one release can be deprecated or change meaning in a following release. Therefore, you should review environment variables that are set for one release, to ensure that they still apply after any upgrade.

For information on compatibility between releases, see the Java on z/OS Web site at <http://www.ibm.com/servers/eserver/zseries/software/java/>.

Private storage usage

The single most common class of failures after a successful install of the SDK are those related to insufficient private storage.

As discussed in detail in “Understanding Memory Usage” on page 161, LE provides storage from Subpool 2, key 8 for C/C++ programs like the JVM that use

C runtime library calls like `malloc()` to obtain memory. The LE HEAP refers to the areas obtained for all C/C++ programs that run in a process address space and request storage.

This area is used for the allocation of the Java heap where instances of Java objects are allocated and managed by Garbage Collection. The area is used also for any underlying allocations that the JVM makes during operations. For example, the JIT compiler obtains work areas for compilation of methods and to store compiled code.

Because the JVM must preallocate the maximum Java heap size so that it is contiguous, the total private area requirement is that of the maximum Java heap size that is set by the **-Xmx** option (or the 64 MB default if this is not set), plus an allowance for underlying allocations. A total private area of 140 MB is therefore a reasonable requirement for an instance of a JVM that has the default maximum heap size.

If the private area is restricted by either a system parameter or user exit, failures to obtain private storage occur. These failures show as `OutOfMemoryErrors` or `Exceptions`, failures to load libraries, or failures to complete subcomponent initialization during startup.

Setting up dumps

The JVM generates a Jav_dump and System Transaction Dump (SYSTDUMP) when particular events occur.

The JVM, by default, generates the dumps when any of the following occurs:

- A `SIGQUIT` signal is received
- The JVM exits because of an error
- An unexpected native exception occurs (for example, a `SIGSEGV`, `SIGILL`, or `SIGFPE` signal is received)

You can use the **-Xdump** option to change the dumps that are produced on the various types of signal and the naming conventions for the dumps. For further details, see Chapter 21, “Using dump agents,” on page 223.

Failing transaction dumps (IEATDUMPS)

If a requested IEATDUMP cannot be produced, the JVM sends a message to the operator console. For example:

```
JVMDMP025I IEATDUMP failed RC=0x00000008 RSN=0x00000022 DSN=ABC.JVM.TDUMP.FUNGE2.D070301.T171813
```

These return codes are fully documented in *z/OS V1R7.0 MVS Authorized Assembler Services Reference, 36.1.10 Return and Reason Codes*. Some common return codes are:

RC=0x00000008 RSN=0x00000022

Dump file name too long.

RC=0x00000008 RSN=0x00000026

Insufficient space for IEATDUMP.

RC=0x00000004

Partial dump taken. Typically, 2 GB size limit reached.

If the IEATDUMP produced is partial because of the 2 GB IEATDUMP size limit you should use this message to trigger an SVC dump. To trigger the SVC dump, use a SLIP trap. For example:

```
SLIP SET,A=SVCD,J=FUNGE*,MSGID=JVMDMP025I,ID=JAVA,SDATA=(ALLPSA,NUC,SQA,RGN,LPA,TRT,SUMDUMP),END
```

Multiple transaction dump (IEATDUMP) files on z/OS version 1.10 or newer

For z/OS version 1.10 or newer, on a 64-bit platform, IEATDUMP files are split into several smaller files if the IEATDUMP exceeds the 2 GB file size limit. Each file is given a sequence number.

If you specify a template for the IEATDUMP file name, append the &DS token to enable multiple dumps. The &DS token is replaced by an ordered sequence number, and must be at the end of the file name. For example, X&DS generates file names in the form X001, X002, X003, and so on.

If you specify a template without the &DS token, .X&DS is appended automatically to the end of your template. If your template is too long to append .X&DS, a message is issued advising that the template pattern is too long and that a default pattern will be used.

If you do not specify a template, the default template is used. The default template is:

```
%uid.JVM.TDUMP.D%y%m%d.T%H%M%S.X&DS
```

You must merge the sequence of IEATDUMP files before IPCS can process the data. To merge the sequence of IEATDUMP files, use the TSO panel **IPCS → Utility → Copy MVS dump dataset**, or the IPCS COPYDUMP command.

For more information, see APAR: OA24232.

Note: For versions of z/OS prior to version 1.10, IEATDUMP file handling is unchanged.

General debugging techniques

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

In addition to the information given in this section, you can obtain z/OS publications from the IBM Web site. Go to <http://www.ibm.com/support/publications/us/library/>, and then choose the documentation link for your platform.

There are several diagnostic tools available with the JVM to help diagnose problems:

- Starting Javadumps, see Chapter 22, “Using Jav_dump,” on page 245.
- Starting Heapdumps, see Chapter 23, “Using Heapdump,” on page 257.
- Starting system dumps, see Chapter 24, “Using system dumps and the dump viewer,” on page 263.

z/OS provides various commands and tools that can be useful in diagnosing problems.

Using IPCS commands

The Interactive Problem Control System (IPCS) is a tool provided in z/OS to help you diagnose software failures. IPCS provides formatting and analysis support for dumps and traces produced by z/OS.

Here are some sample IPCS commands that you might find useful during your debugging sessions. In this case, the address space of interest is ASID(x'7D').

ip verbx ledata 'nthreads(*)'

This command provides the stack traces for the TCBs in the dump.

ip setd asid(x'007d')

This command is to set the default ASID; for example, to set the default asid to x'007d'.

ip verbx ledata 'all,asid(007d),tcb(tttttt)'

In this command, the **all** report formats out key LE control blocks such as CAA, PCB, ZMCH, CIB. In particular, the CIB/ZMCH captures the PSW and GPRs at the time the program check occurred.

ip verbx ledata 'cee,asid(007d),tcb(tttttt)'

This command formats out the traceback for one specific thread.

ip summ regs asid(x'007d')

This command formats out the TCB/RB structure for the address space. It is rarely useful for JVM debugging.

ip verbx sumdump

Then issue **find 'slip regs sa'** to locate the GPRs and PSW at the time a SLIP TRAP is matched. This command is useful for the case where you set a SA (Storage Alter) trap to catch an overlay of storage.

ip omvsdata process detail asid(x'007d')

This command generates a report for the process showing the thread status from a USS kernel perspective.

ip select all

This command generates a list of the address spaces in the system at the time of the dump, so that you can tie up the ASID with the JOBNAME.

ip systrace asid(x'007d') time(gmt)

This command formats out the system trace entries for all threads in this address space. It is useful for diagnosing loops. **time(gmt)** converts the TOD Clock entries in the system trace to a human readable form.

For further information about IPCS, see the z/OS documentation (*z/OS V1R7.0 MVS™ IPCS Commands*).

Using dbx

The dbx utility has been improved for z/OS V1R6. You can use dbx to analyze transaction (or system) dumps and to debug a running application.

For information about dbx, see the z/OS documentation; *z/OS V1R6.0 UNIX System Services Programming Tools* at <http://publibz.boulder.ibm.com/epubs/pdf/bpxza630.pdf>.

Interpreting error message IDs

While working in the OMVS, if you get an error message and want to understand exactly what the error message means there is a Web site you can go to.

Go to: <http://www-03.ibm.com/systems/z/os/zos/bkserv/lookat/index.html> and enter the message ID. Then select your OS level and then press enter. The output will give a better understanding of the error message. To decode the errno2 values, use the following command:

```
bpxmtext <reason_code>
```

Reason_code is specified as 8 hexadecimal characters. Leading zeros can be omitted.

Diagnosing crashes

A crash should occur only because of a fault in the JVM, or because of a fault in native (JNI) code that is being run inside the Java process. A crash is more strictly defined on z/OS as a program check that is handled by z/OS UNIX[®] as a fatal signal (for example, SIGSEGV for PIC4; 10, 11, or SIGILL for PIC1).

Documents to gather

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

When one of these fatal signals occurs, the JVM Signal Handler takes control. The default action of the signal handler is to produce a transaction dump (through the BCP IEATDUMP service), a JVM snap trace dump, and a formatted Javadump. Output should be written to the message stream that is written to stderr in the form of:

```
Unhandled exception
Type=Segmentation error vmState=0x00000000
Target=2_30_20060227_05498_bHdSMr (z/OS 01.06.00)
CPU=s390 (2 logical CPUs) (0x180000000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=00000000 Signal_Code=00000035
Handler1=115F8590 Handler2=116AFC60
gpr0=00000064 gpr1=00000000 gpr2=117A3D70 gpr3=00000000
gpr4=114F5280 gpr5=117C0E28 gpr6=117A2A18 gpr7=9167B460
gpr8=0000007E gpr9=116AF5E8 gpr10=1146E21C gpr11=0000007E
gpr12=1102C7D0 gpr13=11520838 gpr14=115F8590 gpr15=00000000
psw0=078D0400 psw1=917A2A2A
fpr0=48441040 fpr1=3FFF1999 fpr2=4E800001 fpr3=99999999
fpr4=45F42400 fpr5=3FF00000 fpr6=00000000 fpr7=00000000
fpr8=00000000 fpr9=00000000 fpr10=00000000 fpr11=00000000
fpr12=00000000 fpr13=00000000 fpr14=00000000 fpr15=00000000
Program_Unit_Name=
Program_Unit_Address=1167B198 Entry_Name=j9sig_protect
Entry_Address=1167B198
JVMDUMP006I Processing Dump Event "gpf", detail "" - Please Wait.
JVMDUMP007I JVM Requesting System Dump using 'CHAMBER.JVM.TDUMP.CHAMBER1.D060309.T144842'
IEATDUMP in progress with options SDATA=(LPA,GRSQ,LSQA,NUC,PSA,RGN,SQA,SUM,SWA,TRT)
IEATDUMP success for DSN='CHAMBER.JVM.TDUMP.CHAMBER1.D060309.T144842'
JVMDUMP010I System Dump written to CHAMBER.JVM.TDUMP.CHAMBER1.D060309.T144842
JVMDUMP007I JVM Requesting Snap Dump using '/u/chamber/test/ras/Snap0001.20060309.144842.196780.trc'
JVMDUMP010I Snap Dump written to /u/chamber/test/ras/Snap0002.20060309.144842.196780.trc
JVMDUMP007I JVM Requesting Java Dump using '/u/chamber/test/ras/javacore.20060309.144842.196780.txt'
JVMDUMP010I Java Dump written to /u/chamber/test/ras/javacore.20060309.144842.196780.txt
JVMDUMP013I Processed Dump Event "gpf", detail "".
```

The output shows the location in HFS into which the Javadump file was written and the name of the MVS data set to which the transaction dump is written. These locations are configurable and are described in Chapter 20, “Overview of the available diagnostics,” on page 217 and Chapter 21, “Using dump agents,” on page 223.

These documents provide the ability to determine the failing function, and therefore decide which product owns the failing code, be it the JVM, application JNI code, or native libraries acquired from another vendor (for example native JDBC drivers).

The JVM will display error messages if it is unable to produce the dumps. The IEATDUMP error return codes, RC=... and RSN=..., are included in the messages. These return codes are fully documented in *z/OS V1R7.0 MVS Authorized Assembler Services Reference*, 36.1.10 Return and Reason Codes.

This example shows the error messages displayed when there is insufficient disk space to write the IEATDUMP:

```
JVMDUMP007I JVM Requesting System dump using 'J9BUILD.JVM.TDUMP.SSHD1.D080326.T081447'
IEATDUMP in progress with options SDATA=(LPA,GRSQ,LSQA,NUC,PSA,RGN,SQA,SUM,SWA,TRT)
IEATDUMP failure for DSN='J9BUILD.JVM.TDUMP.SSHD1.D080326.T081447' RC=0x00000008 RSN=0x00000026
JVMDUMP012E Error in System dump: J9BUILD.JVM.TDUMP.SSHD1.D080326.T081447
```

z/OS V1R7.0 MVS Authorized Assembler Services Reference is available at <http://www.ibm.com/servers/eserver/zseries/zose/bkserv/r7pdf/mvs.html>.

Determining the failing function

The most practical way to find where the exception occurred is to review either the CEEDUMP or the Javadump. Both of these reports show where the exception occurred and the native stack trace for the failing thread.

The same information can be obtained from the transaction dump by using either the dump viewer (see Chapter 24, “Using system dumps and the dump viewer,” on page 263), the dbx debugger, or the IPCS LEDATA VERB Exit.

The CEEDUMP shows the C-Stack (or native stack, which is separate from the Java stack that is built by the JVM). The C-stack frames are also known on z/OS as Dynamic Storage Areas (DSAs), because a DSA is the name of the control block that LE provides as a native stack frame for a C/C++ program. The following traceback from a CEEDUMP shows where a failure occurred:

Traceback:

DSA	Entry	E Offset	Load Mod	Program Unit	Service	Status
00000001	__cdump	+00000000	CELQLIB		HLE7709	Call
00000002	@@WRAP@MULTHD					
		+00000266	CELQLIB			Call
00000003	j9dump_create					
		+0000035C	*PATHNAM		j040813	Call
00000004	doSystemDump	+0000008C	*PATHNAM		j040813	Call
00000005	triggerDumpAgents					
		+00000270	*PATHNAM		j040813	Call
00000006	vmGPHandler	+00000C4C	*PATHNAM		j040813	Call
00000007	gpHandler	+000000D4	*PATHNAM		j040813	Call
00000008	__zerro	+00000BC4	CELQLIB		HLE7709	Call
00000009	__zerros	+0000016E	CELQLIB		HLE7709	Call
0000000A	CEEHDSP	+00003A2C	CELQLIB	CEEHDSP	HLE7709	Call
0000000B	CEEOSIGJ	+00000956	CELQLIB	CEEOSIGJ	HLE7709	Call
0000000C	CELQHROD	+00000256	CELQLIB	CELQHROD	HLE7709	Call
0000000D	CEEOSIGG	-08B3FBB0	CELQLIB	CEEOSIGG	HLE7709	Call
0000000E	CELQHROD	+00000256	CELQLIB	CELQHROD	HLE7709	Call
0000000F	Java_dumpTest_runTest					
		+00000044	*PATHNAM			Exception
00000010	RUNCALLINMETHOD					
		-0000F004	*PATHNAM			Call
00000011	gpProtectedRunCallInMethod					
		+00000044	*PATHNAM		j040813	Call
00000012	j9gp_protect	+00000028	*PATHNAM		j040813	Call
00000013	gpCheckCallin					
		+00000076	*PATHNAM		j040813	Call
00000014	callStaticVoidMethod					
		+00000098	*PATHNAM		j040813	Call
00000015	main	+000029B2	*PATHNAM		j904081	Call
00000016	CELQINIT	+00001146	CELQLIB	CELQINIT	HLE7709	Call

DSA	DSA Addr	E Addr	PU Addr	PU Offset	Comp Date	Attributes			
00000001	00000001082F78E0	000000001110EB38	0000000000000000	*****	20040312	XPLINK EBCDIC	POSIX	IEEE	
00000002	00000001082F7A20	00000000110AF458	0000000000000000	*****	20040312	XPLINK EBCDIC	POSIX	Float	
00000003	00000001082F7C00	0000000011202988	0000000000000000	*****	20040817	XPLINK EBCDIC	POSIX	IEEE	
00000004	00000001082F8100	0000000011213770	0000000000000000	*****	20040817	XPLINK EBCDIC	POSIX	IEEE	
00000005	00000001082F8200	0000000011219760	0000000000000000	*****	20040817	XPLINK EBCDIC	POSIX	IEEE	
00000006	00000001082F8540	000000007CD4BDA8	0000000000000000	*****	20040817	XPLINK EBCDIC	POSIX	IEEE	
00000007	00000001082F9380	00000000111FF190	0000000000000000	*****	20040817	XPLINK EBCDIC	POSIX	IEEE	
00000008	00000001082F9480	00000000111121E0	0000000000000000	*****	20040312	XPLINK EBCDIC	POSIX	IEEE	
00000009	00000001082FA0C0	0000000011112048	0000000000000000	*****	20040312	XPLINK EBCDIC	POSIX	IEEE	
0000000A	00000001082FA1C0	0000000010DB8EA0	0000000010DB8EA0	00003A2C	20040312	XPLINK EBCDIC	POSIX	Float	
0000000B	00000001082FCAE0	0000000010E3D530	0000000010E3D530	00000956	20040312	XPLINK EBCDIC	POSIX	Float	
0000000C	00000001082FD4E0	0000000010D76778	0000000010D76778	00000256	20040312	XPLINK EBCDIC	POSIX	Float	
0000000D	00000001082FD720	0000000010E36C08	0000000010E36C08	08B3FBB0	20040312	XPLINK EBCDIC	POSIX	Float	
0000000E	00000001082FE540	0000000010D76778	0000000010D76778	00000256	20040312	XPLINK EBCDIC	POSIX	Float	
0000000F	00000001082FE780	00000000122C66B0	0000000000000000	*****	20040802	XPLINK EBCDIC	POSIX	IEEE	
00000010	00000001082FE880	000000007CD28030	0000000000000000	*****	^C"22^04^FF^FDu^58	XPLINK EBCDIC	POSIX	IEEE	
00000011	00000001082FEC80	000000007CD515B8	0000000000000000	*****	20040817	XPLINK EBCDIC	POSIX	IEEE	
00000012	00000001082FED80	00000000111FF948	0000000000000000	*****	20040817	XPLINK EBCDIC	POSIX	IEEE	
00000013	00000001082FEE80	000000007CD531A8	0000000000000000	*****	20040817	XPLINK EBCDIC	POSIX	IEEE	
00000014	00000001082FEF80	000000007CD4F148	0000000000000000	*****	20040817	XPLINK EBCDIC	POSIX	IEEE	

Note:

1. The stack frame that has a status value of Exception indicates where the crash occurred. In this example, the crash occurs in the function Java_dumpTest_runTest.
2. The value under Service for each DSA is the service string. The string is built in the format of jjymmdd, where j is the identifier for the code owner and yymmdd is the build date. A service string with this format indicates that the function is part of the JVM. All functions have the same build date, unless you have been supplied with a dll by IBM Service for diagnostic or temporary fix purposes.

Working with TDUMPs using IPCS

A TDUMP or Transaction Dump is generated from the MVS service IEATDUMP by default in the event of a program check or exception in the JVM. You can disable the generation of a TDUMP, but it is not recommended by IBM Service.

A TDUMP can contain multiple Address Spaces. It is important to work with the correct address space associated with the failing java process.

To work with a TDUMP in IPCS, here is a sample set of steps to add the dump file to the IPCS inventory:

1. Browse the dump data set to check the format and to ensure that the dump is correct.
2. In IPCS option 3 (**Utility Menu**), suboption 4 (**Process list of data set names**) type in the TSO HLQ (for example, DUMPHLQ) and press Enter to list data sets. You must ADD (A in the command-line alongside the relevant data set) the uncompressed (untersted) data set to the IPCS inventory.
3. You can select this dump as the default one to analyze in two ways:
 - In IPCS option 4 (**Inventory Menu**) type SD to add the selected data set name to the default globals.
 - In IPCS option 0 (**DEFAULTS Menu**), change Scope and Source
Scope ==> BOTH (LOCAL, GLOBAL, or BOTH)

```
Source ==> DSNAME('DUMPHLQ.UNTERSED.SIGSEGV.DUMP')
Address Space ==>
Message Routing ==> NOPRINT TERMINAL
Message Control ==> CONFIRM VERIFY FLAG(WARNING)
Display Content ==> NOMACHINE REMARK REQUEST NOSTORAGE SYMBOL
```

If you change the Source default, IPCS displays the current default address space for the new source and ignores any data entered in the address space field.

4. To initialize the dump, select one of the analysis functions, such as IPCS option **2.4 SUMMARY - Address spaces and tasks**, which will display something like the following and give the TCB address. (Note that non-zero CMP entries reflect the termination code.)

```
TCB: 009EC1B0
CMP..... 940C4000 PKF..... 80      LMP..... FF      DSP..... 8C
TSFLG.... 20      STAB..... 009FD420 NDSP..... 00002000
JSCB..... 009ECCB4 BITS..... 00000000 DAR..... 00
RTWA..... 7F8BEDF0 FBYT1.... 08
Task non-dispatchability flags from TCBFLGS5:
Secondary non-dispatchability indicator
Task non-dispatchability flags from TCBNDSP2:
SVC Dump is executing for another task

SVRB: 009FD9A8
WLIC..... 00000000 OPSW..... 070C0000 81035E40
LINK..... 009D1138

PRB: 009D1138
WLIC..... 00040011 OPSW..... 078D1400 B258B108
LINK..... 009ECBF8
EP..... DFSPCJB0 ENTPT.... 80008EF0

PRB: 009ECBF8
WLIC..... 00020006 OPSW..... 078D1000 800091D6
LINK..... 009ECC80
```

Useful IPCS commands and some sample output

Some IPCS commands that you can use when diagnosing crashes.

In IPCS option 6 (**COMMAND Menu**) type in a command and press the Enter key:

ip st

Provides a status report.

ip select all

Shows the Jobname to ASID mapping:

```
ASID JOBNAME  ASCBADDR  SELECTION CRITERIA
-----
0090 H121790  00EFAB80  ALL
0092 BPXAS    00F2E280  ALL
0093 BWASP01  00F2E400  ALL
0094 BWASP03  00F00900  ALL
0095 BWEBP18  00F2EB80  ALL
0096 BPXAS    00F8A880  ALL
```

ip systrace all time(local)

Shows the system trace:

```
PR ASID,WU-ADDR-  IDENT CD/D PSW----- ADDRESS-  UNIQUE-1 UNIQUE-2 UNIQUE-3
                                         UNIQUE-4 UNIQUE-5 UNIQUE-6

09-0094 009DFE88  SVCR   6 078D3400 8DBF7A4E 8AA6C648 0000007A 24AC2408
09-0094 05C04E50  SRB    070C0000 8AA709B8 00000094 02CC90C0 02CC90EC
                                         009DFE88 A0
09-0094 05C04E50  PC     ...  0      0AA70A06          0030B
09-0094 00000000  SSRV  132      00000000 0000E602 00002000 7EF16000
                                         00940000
```

For suspected loops you might need to concentrate on ASID and exclude any branch tracing:

ip systrace asid(x'3c') exclude(br)

ip summ format asid(x'94')

To find the list of TCBs, issue a find command for "T C B".

ip verbx ledata 'ceedump asid(94) tcb(009DFE88)'

Obtains a traceback for the specified TCB.

ip omvsdata process detail asid(x'94')

Shows a USS perspective for each thread.

ip verbx vsmdata 'summary noglobal'

Provides a memory usage report:

LOCAL STORAGE MAP

Extended LSQA/SWA/229/230	80000000 <- Top of Ext. Private 80000000 <- Max Ext. User Region Address 7F4AE000 <- ELSQA Bottom
(Free Extended Storage)	127FE000 <- Ext. User Region Top
Extended User Region	10D00000 <- Ext. User Region Start
:	:
: Extended Global Storage	:
===== <- 16M Line	
: Global Storage	:
:	: A00000 <- Top of Private

LSQA/SWA/229/230	A00000 <- Max User Region Address
	9B8000 <- LSQA Bottom
(Free Storage)	
	7000 <- User Region Top
User Region	
	6000 <- User Region Start
: System Storage :	
: :	0

ip verbx ledata 'nthreads(*)'

Obtains the tracebacks for all threads.

ip status regs

Shows the PSW and registers:

CPU STATUS:

BLS18058I Warnings regarding STRUCTURE(Psa) at ASID(X'0001') 00:

BLS18300I Storage not in dump

PSW=00000000 00000000

(Running in PRIMARY key 0 AMODE 24 DAT OFF)

DISABLED FOR PER I/O EXT MCH

ASCB99 at FA3200 JOB(JAVADV1) for the home ASID

ASXB99 at 8FDD00 and TCB99G at 8C90F8 for the home ASID

HOME ASID: 0063 PRIMARY ASID: 0063 SECONDARY ASID: 0063

General purpose register values

Left halves of all registers contain zeros

0-3 00000000 00000000 00000000 00000000

4-7 00000000 00000000 00000000 00000000

8-11 00000000 00000000 00000000 00000000

12-15 00000000 00000000 00000000 00000000

Access register values

0-3 00000000 00000000 00000000 00000000

4-7 00000000 00000000 00000000 00000000

8-11 00000000 00000000 00000000 00000000

12-15 00000000 00000000 00000000 00000000

Control register values

0-1 00000000_5F04EE50 00000001_FFC3C007

2-3 00000000_5A057800 00000001_00C00063

4-5 00000000_00000063 00000000_048158C0

6-7 00000000_00000000 00000001_FFC3C007

8-9 00000000_00000000 00000000_00000000

10-11 00000000_00000000 00000000_00000000

12-13 00000000_0381829F 00000001_FFC3C007

14-15 00000000_DF884811 00000000_7F5DC138

ip cbf rtct

Helps you to find the ASID by looking at the ASTB mapping to see which ASIDs are captured in the dump.

ip verbx vsmdata 'nog summ'

Provides a summary of the virtual storage management data areas:

DATA FOR SUBPOOL 2 KEY 8 FOLLOWS:

-- DQE LISTING (VIRTUAL BELOW, REAL ANY64)

DQE: ADDR 12C1D000 SIZE 32000

DQE: ADDR 1305D000 SIZE 800000

DQE: ADDR 14270000 SIZE 200000

DQE: ADDR 14470000 SIZE 10002000

DQE: ADDR 24472000 SIZE 403000

DQE: ADDR 24875000 SIZE 403000

```

DQE:  ADDR 24C78000 SIZE 83000
DQE:  ADDR 24CFB000 SIZE 200000
DQE:  ADDR 250FD000 SIZE 39B000
                                FQE: ADDR 25497028 SIZE FD8
DQE:  ADDR 25498000 SIZE 735000
                                FQE: ADDR 25BCC028 SIZE FD8
DQE:  ADDR 25D36000 SIZE 200000
DQE:  ADDR 29897000 SIZE 200000
DQE:  ADDR 2A7F4000 SIZE 200000
DQE:  ADDR 2A9F4000 SIZE 200000
DQE:  ADDR 2AC2F000 SIZE 735000
                                FQE: ADDR 2B363028 SIZE FD8
DQE:  ADDR 2B383000 SIZE 200000
DQE:  ADDR 2B5C7000 SIZE 200000
DQE:  ADDR 2B857000 SIZE 1000

```

***** SUBPOOL 2 KEY 8 TOTAL ALLOC: 132C3000 (00000000 BELOW, 132C3000

ip verbx ledata 'all asid(54) tcb(009FD098)'

Finds the PSW and registers at time of the exception:

```

+000348 MCH_EYE:ZMCH
+000350 MCH_GPR00:00000000 000003E7 MCH_GPR01:00000000 00000000
+000360 MCH_GPR02:00000001 00006160 MCH_GPR03:00000000 00000010
+000370 MCH_GPR04:00000001 082FE780 MCH_GPR05:00000000 000000C0
+000380 MCH_GPR06:00000000 00000000 MCH_GPR07:00000000 127FC6E8
+000390 MCH_GPR08:00000000 00000007 MCH_GPR09:00000000 127FC708
+0003A0 MCH_GPR10:00000001 08377D70 MCH_GPR11:00000001 0C83FB78
+0003B0 MCH_GPR12:00000001 08300C60 MCH_GPR13:00000001 08377D00
+0003C0 MCH_GPR14:00000000 112100D0 MCH_GPR15:00000000 00000000
+0003D0 MCH_PSW:07852401 80000000 00000000 127FC6F8 MCH_ILC:0004
+0003E2 MCH_IC1:00 MCH_IC2:04 MCH_PFT:00000000 00000000
+0003F0 MCH_FLT_0:48410E4F 6C000000 4E800001 31F20A8D
+000400 MCH_FLT_2:406F0000 00000000 00000000 00000000
+000410 MCH_FLT_4:45800000 00000000 3FF00000 00000000
+000420 MCH_FLT_6:00000000 00000000 00000000 00000000
+0004B8 MCH_EXT:00000000 00000000

```

blscddir dsname('DUMPHLQ.ddir')

Creates an IPCS DDIR.

runc addr(2657c9b8) link(20:23) chain(9999) le(x'1c') or runc addr(25429108) structure(tcb)

Runs a chain of control blocks using the RUNCHAIN command.

addr: the start address of the first block

link: the link pointer start and end bytes in the block (decimal)

chain: the maximum number of blocks to be searched (default=999)

le: the length of data from the start of each block to be displayed (hex)

structure: control block type

Debugging hangs

A hang refers to a process that is still present, but has become unresponsive.

This lack of response can be caused by any one of these reasons:

- The process has become deadlocked, so no work is being done. Usually, the process is taking up no CPU time.
- The process has become caught in an infinite loop. Usually, the process is taking up high CPU time.
- The process is running, but is suffering from very bad performance. This is not an actual hang, but is often initially mistaken for one.

The process is deadlocked

A deadlocked process does not use any CPU time.

You can monitor this condition by using the USS **ps** command against the Java process:

UID	PID	PPID	C	STIME	TTY	TIME	CMD
CBAILEY	253	743	-	10:24:19	ttyp0003	2:34	java -classpath .Test2Frame

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

For an explanation of deadlocks and how the Javdump tool is used to diagnose them, see “Locks, monitors, and deadlocks (LOCKS)” on page 252.

The process is looping

If no deadlock exists between threads and the process appears to be hanging but is consuming CPU time, look at the work the threads are doing. To do this, take a console-initiated dump (SVC dump).

Follow these steps to take a console-initiated dump:

1. Use the operating system commands (**D OMVS,A=ALL**) or **SDSF (DA = Display Active)** to locate the ASID of interest.
2. Use the **DUMP** command to take a console-initiated dump both for hangs and for loops:

```
DUMP COMM=(Dump for problem 12345)
R xx,ASID=(53,d),DSPNAME=('OMVS '.*),CONT
R yy,SDATA=(GRSQ,LSQA,RGN,SUM,SWA,TRT,LPA,NUC,SQA)
```

Prefix all commands on the SDSF panels with '/'. The console responds to the **DUMP** command with a message requesting additional 'operand(s)', and provides you with a 2-digit reply ID. You supply the additional operands using the R (reply) command, specifying the reply ID (shown as 'xx' or 'yy' in the example above). You can use multiple replies for the operands by specifying the CONT operand, as in the example above.

You can select the process to dump using the z/OS job name instead of the ASID:

```
R xx,JOBNAME=SSHD9,CONT
```

When the console dump has been generated, you can view the Systrace in IPCS to identify threads that are looping. You can do this in IPCS as follows:

```
ip systrace asid(x'007d') time(gmt)
```

This command formats out the system trace entries for all threads that are in address space 0x7d. The time(gmt) option converts the TOD clock entries, which are in the system trace, to a human readable form.

From the output produced, you can determine which are the looping threads by identifying patterns of repeated CLCK and EXT1005 interrupt trace entries, and subsequent redispach DSP entries. You can identify the instruction address range of the loop from the PSWs (Program Status Words) that are traced in these entries.

You can also analyze z/OS console (SVC) dumps using the system dump viewer provided in the SDK, see Chapter 24, “Using system dumps and the dump viewer,” on page 263.

The process is performing badly

If you have no evidence of a deadlock or an infinite loop, the process is probably suffering from very bad performance. Bad performance can be caused because threads have been placed into explicit sleep calls, or by excessive lock contention, long garbage collection cycles, or for several other reasons. This condition is not a hang and should be handled as a performance problem.

See “Debugging performance problems” on page 163 for more information.

Understanding Memory Usage

To debug memory leaks you need to understand the mechanisms that can cause memory problems, how the JVM uses the LE HEAP, how the JVM uses z/OS virtual storage, and the possible causes of a `java.lang.OutOfMemoryError` exception.

Memory problems can occur in the Java process through two mechanisms:

- A native (C/C++) memory leak that causes increased usage of the LE HEAP, which can be seen as excessive usage of Subpool2, Key 8, or storage, and an excessive Working Set Size of the process address space
- A Java object leak in the Java-managed heap. The leak is caused by programming errors in the application or the middleware. These object leaks cause an increase in the amount of live data that remains after a garbage collection cycle has been completed.

Allocations to LE HEAP

The Java process makes two distinct allocation types to the LE HEAP.

The first type is the allocation of the Java heap that garbage collection manages. The Java heap is allocated during JVM startup as a contiguous area of memory. Its size is that of the maximum Java heap size parameter. Even if the minimum, initial, heap size is much smaller, you must allocate for the maximum heap size to ensure that one contiguous area will be available should heap expansion occur.

The second type of allocation to the LE HEAP is that of calls to `malloc()` by the JVM, or by any native JNI code that is running under that Java process. This includes application JNI code, and vendor-supplied native libraries; for example, JDBC drivers.

z/OS virtual storage

To debug these problems, you must understand how C/C++ programs, such as the JVM, use virtual storage on z/OS. To do this, you need some background understanding of the z/OS Virtual Storage Management component and LE.

The process address space on 31-bit z/OS has 31-bit addressing that allows the addressing of 2 GB of virtual storage. The process address space on 64-bit z/OS has 64-bit addressing that allows the addressing of over 2 GB of virtual storage. This storage includes areas that are defined as common (addressable by code running in all address spaces) and other areas that are private (addressable by code running in that address space only).

The size of common areas is defined by several system parameters and the number of load modules that are loaded into these common areas. On many typical systems, the total private area available is about 1.4 GB. From this area, memory resources required by the JVM and its subcomponents such as the JIT are allocated by calls to `malloc()`. These resources include the Java heap and memory required by application JNI code and third-party native libraries.

A Java `OutOfMemoryError` exception typically occurs when the Java heap is exhausted. For further information on z/OS storage allocation, see: <http://www.redbooks.ibm.com/redbooks/SG247035/>. It is possible for a 31-bit JVM to deplete the private storage area, resulting in an `OutOfMemoryError` exception. For more information, see: “`OutOfMemoryError` exceptions.”

OutOfMemoryError exceptions

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

The operation of the JVM's Garbage Collector is such that objects are continuously allocated on the heap by mutator (application) threads until an object allocation fails. At this point, a garbage collection cycle begins. At the end of the cycle, the allocation is tried again. If successful, the mutator threads resume where they stopped. If the allocation request cannot be fulfilled, an out-of-memory exception occurs. See Chapter 2, “Memory management,” on page 7 for more detailed information.

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

```
java -Xmx320m MyApplication
```

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

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

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

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

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

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

The **-verbose:gc** (**-verbose:gc**) switch causes the JVM to print out messages when a garbage collection cycle begins and ends. These messages indicate how much live data remains on the heap at the end of a collection cycle. In the case of a Java object leak, the amount of free space on the heap after a garbage collection cycle decreases over time. See “-verbose:gc logging” on page 330.

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

The output from a dump can be processed by the FindRoots package to produce a reference tree to point to any mismanaged data collections. See “General debugging techniques” on page 151 above.

If an OutOfMemoryError exception is thrown due to private storage area exhaustion under the 31-bit JVM, verify if the environment variable `_BPX_SHAREAS` is set to NO. If `_BPX_SHAREAS` is set to YES multiple processes are allowed to share the same virtual storage (address space). The result is a much quicker depletion of private storage area. For more information on `_BPX_SHAREAS`, see <http://publib.boulder.ibm.com/infocenter/zos/v1r10/topic/com.ibm.zos.r10.bpxb200/shbene.htm>.

Debugging performance problems

Locating the causes of poor performance is often difficult. Although many factors can affect performance, the overall effect is generally perceived as poor response or slow execution of your program.

Correcting one performance problem might cause more problems in another area. By finding and correcting a bottleneck in one place you might only shift the cause of poor performance to other areas. To improve performance, experiment with tuning different parameters, monitoring the effect, and retuning until you are satisfied that your system is performing acceptably

Finding the bottleneck

The aspects of the system that you are most interested in measuring are CPU usage and memory usage. It is possible that even after extensive tuning efforts the CPU is not powerful enough to handle the workload, in which case a CPU upgrade is required. Similarly, if the program is running in an environment in which it does not have enough memory after tuning, you must increase memory size.

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

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

z/OS systems resource usage

The z/OS Resource Measurement Facility (RMF™) gives a detailed view of z/OS processor, memory, and I/O device performance metrics.

JVM heap sizing

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

The Java heap size is one of the most important tuning parameters of your JVM. See “How to do heap sizing” on page 21 for information on how to correctly set the size of your heap.

JIT compilation and performance

The JIT is another area that can affect the performance of your program. When deciding whether or not to use JIT compilation, you must make a balance between faster execution and increased processor usage during compilation .

The performance of short-running applications can be improved by using the **-Xquickstart** command-line parameter. The JIT is switched on by default, but you can use **-Xint** to turn it off. You also have considerable flexibility in controlling JIT processing. For more details about the JIT, see Chapter 5, “The JIT compiler,” on page 35 and Chapter 26, “JIT problem determination,” on page 317.

Application profiling

You can learn a lot about your Java application by using the hprof profiling agent. Statistics about CPU and memory usage are presented along with many other options.

The hprof tool is discussed in detail in Chapter 32, “Using the HPROF Profiler,” on page 385. **-Xrunhprof:help** gives you a list of suboptions that you can use with hprof.

MustGather information for z/OS

The more information that you can collect about a problem, the easier it is to diagnose that problem. A large set of data can be collected, although some is relevant to particular problems.

The data collected from a fault situation in z/OS depends on the problem symptoms, but could include some or all of the following:

- Transaction dump - an unformatted dump requested by the MVS BCP IEATDUMP service. This dump can be post-processed with the dump viewer (see Chapter 24, “Using system dumps and the dump viewer,” on page 263), the dbx debugger, or IPCS (Interactive Problem Control System). This is part of the system dump.
- CEEDUMP - formatted application level dump, requested by the cdump system call.
- Javadump - formatted internal state data produced by the IBM Virtual Machine for Java.
- Binary or formatted trace data from the JVM internal high performance trace. See “Using method trace” on page 309 and Chapter 25, “Tracing Java applications and the JVM,” on page 283.

- Debugging messages written to stderr (for example, the output from the JVM when switches like **-verbose:gc**, **-verbose**, or **-Xtgc** are used).
- Java stack traces when exceptions are thrown.
- Other unformatted system dumps obtained from middleware products or components (for example, SVC dumps requested by WebSphere for z/OS).
- SVC dumps obtained by the MVS Console DUMP command (typically for loops or hangs, or when the IEATDUMP fails).
- Trace data from other products or components (for example LE traces or the Component trace for z/OS UNIX).
- Heapdumps, if generated automatically, are required for problem determination. You should also take a Heapdump if you have a memory or performance problem.

Chapter 14. IBM i problem determination

This section describes problem determination for the IBM Technology for Java VM on i5/OS versions V5R4 and higher.

Note: IBM i is the integrated operating environment formerly referred to as IBM i5/OS®. The documentation might refer to IBM i as i5/OS.

The 32-bit version of the VM technology described in this set of topics was first made available on the System i™ family in V5R4 and is not the default VM implementation for Java in that release. The default VM for Java on System i® is the so-called "Classic VM" – a 64-bit VM for Java that has been part of i5/OS and OS/400® since V4R2.

This set of topics describes problem determination on i5/OS in these sections:

- "Determining which VM is in use"
- "Setting up your IBM Technology for Java Environment"
- "General debugging techniques" on page 173
- "Debugging performance problems" on page 175
- "Diagnosing crashes" on page 182
- "Diagnosing hangs" on page 182
- "Understanding memory usage" on page 183
- "Using dbx" on page 186

Determining which VM is in use

Before diagnosing any Java-related problems on i5/OS, you must determine which VM is involved in the problem. In some cases, it might be worthwhile attempting to reproduce a given problem on the "other" VM, whichever that might be. Identical failures between the two different VMs suggest very strongly that the application is at fault. Alternatively, if the VMs are behaving differently, including unexpected success on either of them, one of the two VM implementations might have an internal problem.

The most direct way to determine which VM was involved in a problem - either after the VM process has terminated or while a VM process is still available - is to find and display the i5/OS joblog for the VM process, where there is an identifying message written by the system when a VM for Java is created.

The joblog of each i5/OS job that creates a VM contains a JVAB56D message identifying the VM involved; either Classic VM or IBM Technology for Java. Use the rest of this section to perform problem determination only if the VM reported in this message is IBM Technology for Java.

Setting up your IBM Technology for Java Environment

This section describes how to configure your environment to run the IBM Technology for Java.

Required Software and Licensing

To run with the IBM Technology for Java 5.0, you must be running on i5/OS V5R4 and have the licensed program 5722-JV1 (IBM Developer Kit for Java) with option 8 (J2SE 5.0 32-bit) installed on the partition running the VM. This licensed program and option are shipped on the system CDs included for V5R4.

Ensure the latest fixes are installed on your system. The following table contains the required fixes:

i5/OS (V5R4 or later)	A license of i5/OS is included with every new System i server. If an upgrade from a previous release of OS/400 or i5/OS is needed, contact IBM or the appropriate IBM Business Partner.
Latest i5/OS group HIPER PTFs	<p>If IBM support is available, send a request for the latest i5/OS group HIPER PTFs using the IBM fixes Web site. (http://www.ibm.com/servers/eserver/support/series/fixes/index.html)</p> <p>If current support is not through IBM, contact the appropriate IBM Business Partner.</p> <p>SF99539: 540 Group Hiper</p>
Latest CUM package	CUM packages can be requested using the IBM fixes web site. Use this site to browse and order the latest CUM package.
Latest Java group PTFs	<p>Java support is included with i5/OS. To access the latest Java group PTFs, follow the same ordering process as described above.</p> <p>SF99291: 540 Java</p>
Latest DB2 UDB group PTFs	<p>DB2 Universal Database™ (UDB) for iSeries products is included with i5/OS. To access the latest DB2 UDB for iSeries group PTFs, follow the same ordering process as described above.</p> <p>SF99504: 540 DB2 UDB for iSeries</p>

Configuring JAVA_HOME

The selection of the VM to use depends on the setting of the **JAVA_HOME** environment variable.

You start Java applications using a variety of methods:

1. From QCMD using CL commands RUNJVA and JAVA.
2. From QShell, Qp2term, or Qp2shell using the java command.
3. From a native application using the Java Native Interface

The selection of the VM to use for any or all of these invocation methods depends solely on the setting of the **JAVA_HOME** environment variable when the Java invocation is encountered. See “Setting environment variables for i5/OS PASE or QShell” on page 170 for details on setting i5/OS PASE environment variables.

- If **JAVA_HOME** is not set, or set to the empty string, all invocation methods will use the i5/OS default “Classic VM” implementation. If the Classic VM is not installed, IBM Technology for Java will be used.
- If **JAVA_HOME** is set to a valid IBM Technology for Java VM installation directory, all Java invocation methods will use the specified VM.

- If **JAVA_HOME** is set to any other value, Java invocation will fail with an error message.

When using the default installation directory, specify that IBM Technology for Java should be used by setting **JAVA_HOME** to /QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit. For example:

```
ADDENVVAR ENVVAR(JAVA_HOME)
VALUE('/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit')
```

Or from inside QShell or Qp2term, type:

```
export JAVA_HOME=/QOpenSys/QIBM/ProdData/JavaVM/jdk50/32bit
```

To disable the IBM Technology for Java VM from the CL command line, type:

```
RMVENVVAR JAVA_HOME
```

Or from inside QShell or Qp2term, type:

```
unset JAVA_HOME
```

Enabling i5/OS PASE core files

When a failure occurs, the most important diagnostic data to obtain is the process core file. The VM and i5/OS settings are suitable by default, but, if the process core file is not being correctly generated when an error occurs, the following settings can be checked to verify that they have not been changed from their default values.

Operating system settings

To obtain full core files there are ulimit options that must be set.

To obtain full core files, set the following ulimit options (which are the defaults on i5/OS):

ulimit -c unlimited

Enable corefiles with unlimited size

ulimit -n unlimited

Allows an unlimited number of open file descriptors

ulimit -d unlimited

Sets the user data limit to unlimited

ulimit -f unlimited

Sets the file limit to unlimited

You can display the current ulimit settings with:

```
ulimit -a
```

These values are the “soft” limit, and are applied for each user. These values cannot exceed the “hard” limit value. To display and change the “hard” limits, run the same ulimit commands using the additional -H flag.

The ulimit -c value for the soft limit is ignored and the hard limit value is used so that the core file is generated.

Java Virtual Machine settings

The VM settings should be in place by default, but you can check these settings.

To check that the VM is set to produce a core file when a failure occurs, run the following:

```
java -Xdump:what
```

The output from this command should include the following:

```
dumpFn=doSystemDump
events=gpf+abort
filter=
label=/u/cbailey/sdk/jre/bin/core.%Y%m%d.%H%M%S.%pid.dmp
range=1..0
priority=999
request=serial
opts=
```

events=gpf indicates the behavior when a failure occurs and dumpFn=doSystemDump indicates that a core file should be generated in this case. Other entries in **-Xdump:what** define the behavior for other events. You can change and set options using the command-line option **-Xdump**, which is described in Chapter 21, “Using dump agents,” on page 223.

Available disk space

You must ensure that the disk space available is sufficient for the core file to be written to it.

The location of the core file is determined according to the settings described in “Advanced control of dump agents” on page 233. For the core file to be written to this location, you must have sufficient disk space (up to 4 GB might be required), and the correct permissions for the Java process to write to that location.

Setting environment variables for i5/OS PASE or QShell

You can set environment variables for use in an i5/OS PASE shell or QShell in several ways.

The methods are:

- You can set environment variables “ahead of time,” at the Command Entry panel, before starting the shell. Environment variables, once set, are passed to all child programs and processes. The i5/OS commands for manipulating environment variables are:

WRKENVVAR

Work with environment variables

ADDENVVAR

Add an environment variable

RMENVVAR

Remove an environment variable

CHGENVVAR

Change an environment variable

•

You can wait until you have entered the shell before setting the variables. To set the value of an environment variable in the shell, use the export command. For instance, to set **MYVAR** to ‘myvalue’ you can use:

```
$ export MYVAR=myvalue
```

Note: Qp2term supports several shells, including ksh and csh. The syntax for setting environment variables might differ by shell, but the default Qp2term shell is ksh. QShell is a shell and scripts written in QShell syntax might be used from either shell.

- You can set environment variables (or run any shell command) automatically when a user starts a shell. To do this, add the appropriate commands to a .profile file stored in a user's home directory. (See “Determining the home directory for a user.”)

-

You can also set any number of environment variables, specified in a file of export commands, by “sourcing” or “dotting” that file. This process effectively reads the file line-by-line and executes each line as if it had been entered at the current command prompt.

For example, to source the file “/some/path/to/myfile” of shell commands, you use a single period (or ‘dot’) followed by the name of the file containing the shell commands.

```
$ . /some/path/to/myfile
```

Determining the home directory for a user

You can determine the home directory for a user using the DSPUSRPRF command from i5/OS Command Entry or by using a shell command.

The shell command is:

```
$ system DSPUSRPRF <user> | grep Home
```

Note: The system command provides a simple way to call many i5/OS commands. Output from the commands is written to the standard input stream for the current shell. The example command runs the DSPUSRPRF <user> command, then uses the grep command to display the lines that include the exact string “Home”.

For example:

```
$ system dspusrprf blair | grep Home
Home directory . . . . . : /home/blair
```

Note: Any files used or executed by the i5/OS PASE shell must be encoded in the appropriate LOCALE and have the correct line delimiters. These are typically ASCII encoded files, with a linefeed-only delimiter; see <http://www.ibm.com/systems/power/software/aix/resources.html>. To be able to use the same script from either an i5/OS PASE shell or QShell, the script file in the IFS must be assigned its matching code page. Do this using the setccsid command.

For additional information about working in the QShell environment, see “QShell” in the i5/OS Information Center documentation at <http://publib.boulder.ibm.com/infocenter/iseres/v5r4/topic/rzahz/intro.htm>

For additional information about working in the i5/OS PASE environment, see “i5/OS PASE shells and utilities” in the i5/OS Information Center documentation at: <http://publib.boulder.ibm.com/infocenter/iseres/v5r4/topic/rzalc/pase.htm>. Documentation on i5/OS PASE itself can also be found in the Information Center at <http://publib.boulder.ibm.com/infocenter/iseres/v5r4/topic/rzalf/rzalfintro.htm>.

Setting default Java command-line options

There are several ways to set Java command-line options.

Most methods work on all platforms, and are listed in “Specifying command-line options” on page 439. On i5/OS, one additional option is also available, the `SystemDefault.properties` file. This file is a standard Java properties file, where each line specifies one Java system property or command-line option.

The VM will search the following locations for a `SystemDefault.properties` file. Only the first file found is used. Options specified on the command line will override any options specified in a `SystemDefault.properties` file, but the options specified in this file will override those specified using any other mechanism.

1. A file specified using the **os400.property.file** property passed on the command line to the java command, or through the `JNI_CreateJavaVM` interface.
2. A file specified in the **QIBM_JAVA_PROPERTIES_FILE** environment variable.
3. A `SystemDefault.properties` file found in the user's home directory, as specified using the **user.home** property. By default, this will match the home directory for the user profile running the job; however, the **user.home** property can be overwritten.
4. A `SystemDefault.properties` file found in the `/QIBM/UserData/Java400` directory.

Note: In the first two cases, the properties file might not be named `SystemDefault.properties`.

A line in the properties file that begins with '#' is treated as a comment. By default, any other line in the file is treated as a Java system property. For example, the following file is equivalent to passing the options `"-Dprop1=12345 -Dprop2"` on the java command -line:

```
# Example property file 1
prop1=12345
prop2
```

If the first line of the properties file starts with `"#AllowOptions"`, the processing is changed so that any line beginning with a '-' is treated as a command-line option, rather than as a Java system property. Lines that do not begin with '-' are treated as system properties. For example:

```
#AllowOptions
# Example property file 2
prop1=12345
-Dprop2
-Dprop3=abcd
-Xmx200m
-Xnojit
```

Use of this file will result in:

- The property **prop1** being set to "12345"
- The property **prop2** being set with no value
- The property **prop3** being set to "abcd"
- The maximum heap size being set to 200 MB
- The JIT being unavailable

This processing is provided for compatibility with the i5/OS Classic VM. The Classic VM also processes `SystemDefault.properties` files, but does not use the `"#AllowOptions"` syntax. Therefore, all non-comment lines in the file will be treated as properties, even if they begin with `'.'`.

For example, in the Classic VM, using the second example file above will result in:

- The property **prop1** being set to "12345"
- The property **-Dprop2** being set with no value
- The property **-Dprop3** being set to "abcd"
- The property **-Xmx200m** being set with no value
- The property **-Xnojit** being set with no value

Because properties with names starting with `'.'` are not likely to have any special meaning to the VM, the same `SystemDefault.properties` file can be used for both the Classic and IBM Technology VM while specifying some IBM Technology for Java command-line options that are not available on the Classic VM.

General debugging techniques

A short guide to the diagnostic tools provided by the JVM that can be useful when diagnosing problems with applications running in the IBM Technology for Java.

Additional information can be found in the IBM i information center (<http://publib.boulder.ibm.com/infocenter/iseri/v7r1m0/index.jsp>). Performance information can be found in the "Resource Library" on the Performance Management Web site (<http://www.ibm.com/systems/i/advantages/perfmgmt/index.html>). This information applies to the 32-bit IBM Technology for Java only and not to the 64-bit "Classic" Virtual Machine.

There are several diagnostic tools available with the JVM to help diagnose problems:

- Starting Javadumps, see Chapter 22, "Using Javacore," on page 245.
- Starting Heapdumps, see Chapter 23, "Using Heapdump," on page 257.
- Starting system dumps, see Chapter 24, "Using system dumps and the dump viewer," on page 263.

Diagnosing problems at the command line

Many of the standard commands provided as part of the SDK for Java, (for example: `javac`, `javah`, `rmiserver`) are supported from either an i5/OS PASE shell command prompt or a QShell command prompt. The situation is similar for diagnostic commands designed to work with the IBM Technology for Java virtual machine.

To configure a command-line shell (QShell or i5/OS PASE) suitable for using diagnostic commands provided with the SDK, you must:

1. Start the shell as a user with sufficient system-wide authorities to control and monitor other processes as necessary and to read files anywhere in the target application's IFS directories.
2. Configure the **JAVA_HOME** environment variable.
3. (For tools requiring a GUI.) Configure the **DISPLAY** environment variable that refers to a running X server.

The X server can run on a remote system or directly on the i5/OS partition. Additional configuration (such as using "xhost + <system name>" on the X server system) might also be necessary, depending on your environment. For detailed instructions on setting up an X server in i5/OS, see "Native Abstract Windowing Toolkit" at <http://publib.boulder.ibm.com/infocenter/iseri5/v5r4/topic/rzaha/nawt.htm>.

IBM i debugging commands

Use standard i5/OS system management commands and interfaces to monitor IBM Technology for Java jobs. Some of the more common commands are described in later sections.

Work with Active Jobs (WRKACTJOB)

Provides a list of jobs running on the system with basic resource utilization information for each job. Also provides easy access to WRKJOB.

Work with Job (WRKJOB)

Provides information on a specific job.

Some WRKJOB options include:

"4. Work with spooled files"

Shows the spooled files that were generated by this job. For Java jobs running in batch, any output from the job (using System.out or System.err) is normally written to a spooled file.

"10. Display job log"

Shows the joblog for the current job. Can be used to show the current stack for a specific thread. The stack displayed here does not include Java methods. To see the Java stack for a thread, generate a Javacore; see Chapter 22, "Using Javacore," on page 245.

"20. Work with threads"

Shows the current state of each thread with the total CPU consumed by that thread.

Work with System Status (WRKSYSSTS)

Provides overall system CPU utilization and memory pool details. Use this screen to ensure that non-database paging rates are low, because it is important that the Java heap can be contained in memory without paging.

This screen can also be used to verify that there are no thread transitions to the Ineligible state (from either the Active or the Wait state). If threads are moving to the Ineligible state for a pool in which Java is used, they indicate that the activity level (Max Active) for the pool should be increased.

Work with Disk Status (WRKDSKSTS)

Provides disk activity information for all of the disks in this partition. High disk utilization can degrade performance. Also used to find failed or degraded disk units.

Process Status (ps)

Provides the status of processes (jobs) running on the system. The ps utility is available from either QShell (QSH) or in the i5/OS PASE environment (QP2TERM or QP2SHELL). The available parameters and output of ps in these two environments are different.

For example, in qsh, `ps -u gichora` shows information about all processes for user "gichora":

PID	DEVICE	TIME	FUNCTION	STATUS	JOBID
84	qpadev0004	00:00	cmd-qsh	deqw	026183/gichora/qpadev0004
110	qpadev0006	00:00	cmd-qsh	deqw	026258/gichora/qpadev0006
85	-	00:00	pgm-qzshsh	timw	026184/gichora/qzshsh
1	qpadev0003	00:00	cmd-telnet	selw	026231/gichora/qpadev0003
111	-	00:00	pgm-qzshsh	evtw	026259/gichora/qzshsh
1	qpadev0006	00:00	grp-qezgrp2	grp	026264/gichora/qpadev0006
117	-	00:00	pgm-qzshchld	evtw	026266/gichora/qzshchld
120	-	05:56	-	thdw	026269/gichora/qzshchld

For more information about QShell ps, see the i5/OS information center section on QShell utilities.

The same command works in QP2TERM; however, only i5/OS PASE processes (including IBM Technology for Java VMs) are shown. The output is as follows:

UID	PID	TTY	TIME	CMD
159	120	-	22:48	jvmStartPase

In the i5/OS PASE version of ps, you can use ps to show information about the individual threads in a job using the following parameters:

```
ps -mp PID -o THREAD
```

For more information on other options available in the i5/OS PASE version of ps, see the description given in "AIX debugging commands" on page 90. The i5/OS PASE version of ps does not report CPU utilization as the AIX version does.

Debugger (dbx)

dbx is the AIX standard command-line debugger, used for debugging i5/OS PASE jobs (including the IBM Technology for Java VM).

For additional information, see "Using dbx" on page 186.

Debugging performance problems

Performance problems are difficult to identify in development and test environments. As a result, many production sites experience inexplicable performance problems. Tracking the cause of performance problems can be difficult because there are many possible causes; for example, the network, the hard drives, database time, or poor scalability in software.

Many tools are available to analyze performance. The right tool depends on the type of performance problem being experienced. This section illustrates some of the most common performance analysis tools used on i5/OS dealing with three main potential resource constraints:

- CPU
- Memory
- Input/Output

Analyzing CPU bottlenecks

This section describes tools that can be used to analyze CPU bottlenecks.

Work with Active Job (WRKACTJOB)

This tool shows the CPU used by each job and threads on each job. This information allows you to determine which jobs are responsible for high CPU usage.

The illustration shows that under subsystem QINTER there is a SPECjbb job running and using 61.1% of the CPU. Total CPU being used is 64.1% (seen in the top-left corner), meaning that about 3% must be used by other jobs in the system.

Sample output for WRKACTJOB

```
Work with Active Jobs                                ROMAN
                                                    05/01/06 09:06:06
CPU %:    64.1    Elapsed time:  00:00:00    Active jobs:  254

Type options, press Enter.
 2=Change  3=Hold  4=End  5=Work with  6=Release  7=Display message
 8=Work with spooled files 13=Disconnect ...

Current
Opt Subsystem/Job User      Type CPU % Function      Status
   QBATCH        QSYS      SBS   .0
   QCMN          QSYS      SBS   .0
   QCTL          QSYS      SBS   .0
   QSYSSCD       QPGMR      BCH   .0 PGM-QEZSCNEP  EVTW
   QINTER        QSYS      SBS   .0
   QPADEV0004    QSECOFR    INT   1.2 CMD-AJ      RUN
   QP0ZSPWP     QSECOFR    BCI   .0 PGM-QZSHCHLD  EVTW
   QP0ZSPWP     QSECOFR    BCI  61.1 JVM-spec.jbb.J  THDW
   QZSHSH       QSECOFR    BCI   .0 PGM-QZSHSH  EVTW

More...

Parameters or command
====>
F3=Exit  F5=Refresh  F7=Find  F10=Restart statistics
F11=Display elapsed data F12=Cancel F23=More options F24=More keys
```

Work with System Activity (WRKSYSACT)

This tool shows CPU use for the most active threads and tasks running on the system. For each thread, the priority, CPU use, and various I/O statistics are shown. WRKSYSACT can refresh automatically.

Sample output for WRKSYSACT


```

Work with System Activity
05/01/06 09:12:40 ROMAN
Automatic refresh in seconds . . . . . 5
Elapsed time . . . . . : 00:00:02 Average CPU util . . . . . : 100.0
Number of CPUs . . . . . : 2 Maximum CPU util . . . . . : 100.0
Overall DB CPU util . . . . . : .0 Minimum CPU util . . . . . : 100.0
Current processing capacity: 2.00

Authorization Type . :
Type options, press Enter.
1=Monitor job 5=Work with job

Opt Job or User Number Thread Pty CPU Total Sync Total Async DB
Task Util I/O I/O Util
QP0ZSPWP QSECOFR 094188 00000021 26 24.9 0 0 .0
QP0ZSPWP QSECOFR 094188 0000001E 26 24.9 0 0 .0
QP0ZSPWP QSECOFR 094188 00000020 26 19.8 0 0 .0
QP0ZSPWP QSECOFR 094188 00000022 26 18.8 0 0 .0
QP0ZSPWP QSECOFR 094188 0000001F 36 11.5 0 0 .0
QPADEV0004 QSECOFR 094183 0000000B 1 .1 7 0 .0

Bottom
F3=Exit F10=Update list F11=View 2 F12=Cancel F19=Automatic refresh
F24=More keys
(C) COPYRIGHT IBM CORP. 1981, 2005.

```

This tool is shipped as part of the 5722PT1 LPP (Licensed Program Product).

Performance Explorer

This tool performs detailed analysis of CPU bottlenecks. Performance Explorer (PEX) can collect a variety of performance data about specific applications, programs, or system resources. PEX can collect many different types of data and the complexity of analyzing this data varies.

The following example shows how to analyze a CPU profile.

1. Create a PEX definition. There are several variations, but a typical definition is:
ADDPEXDFN DFN(TPROF5) TYPE(*PROFILE) PRFTYPE(*JOB) JOB(*ALL)
TASK(*ALL) MAXSTG(100000) INTERVAL(5)

Create this definition only once for each system.

2. Collect the PEX data. Start your workload and then run the following commands:
 - a. STRPEX SSNID(*session*) DFN(TPROF5)
 - b. Wait as your application runs and data is collected. 5-10 minutes is usually reasonable.
 - c. ENDPLEX SSNID(*session*)

The *session* variable can be any name of your choice, used for your own reference.

3. Process the data. The two most common tools are:
 - Performance Trace Data Visualizer (PTDV) for System i5® is a tool for processing, analyzing, and viewing Performance Explorer Collection data residing in PEX database files. PTDV provides a graphical view of PEX profile data.
PTDV can be downloaded from <http://www.alphaworks.ibm.com/tech/ptdv>.
 -

IBM Performance Tools for iSeries (LPP 5722-PT1) is a collection of tools and commands that allows you to analyze performance data using views, reports and graphs. You can examine a CPU Profile using the CL Print PEX report (PRTPEXRPT) command.

For example, the following command can be used to generate a report for the CPU profile example above, summarized to show the procedures that were most frequently executed:

```
PRTPEXRPT MBR(session) TYPE(*PROFILE) PROFILEOPT(*SAMPLECOUNT *PROCEDURE)
```

For more information on TPROF data on i5/OS, see <http://www.ibm.com/servers/enable/site/education/wp/9a1a/index.html>.

Analyzing memory problems

This section describes tools that can be used to analyze memory problems.

Work with System Status (WRKSYSSTS)

In i5/OS, main storage can be divided into logical allocations called memory pools. Memory pools can be private, shared, or special shared. The Work with System Status (WRKSYSSTS) command shows information about the current status of the system.

This command displays:

- The number of jobs currently in the system
- The total capacity of the system auxiliary storage pool (ASP)
- The percentage of the system ASP currently in use
- The amount of temporary storage currently in use
- The percentage of system addresses used
- Statistical information related to each storage pool that currently has main storage allocated to it

Sample output for WRKSYSSTS ASTLV(*INTERMED)

```

                                Work with System Status                                SE520B2
                                                                04/28/06 16:48:24
% CPU used . . . . . :      27.9   Auxiliary storage:
% DB capability . . . . :        .0   System ASP . . . . . :    351.6 G
Elapsed time . . . . . : 00:00:01   % system ASP used . . . :    74.7890
Jobs in system . . . . . :    1187   Total . . . . . :        351.6 G
% perm addresses . . . . :    .010   Current unprotect used :    8248 M
% temp addresses . . . . :    .028   Maximum unprotect . . . :    9678 M

```

Type changes (if allowed), press Enter.

System Pool	Pool Size (M)	Reserved Size (M)	Max Active	-----DB----- Fault	Pages	---Non-DB--- Fault	Pages
1	389.62	212.87	+++++	.0	.0	.0	.0
2	6369.87	4.69	903	.0	.0	.0	.0
3	2460.75	.00	188	.0	.0	.9	.9
4	85.37	.00	5	.0	.0	.0	.0

Bottom

Command

====>

F3=Exit F4=Prompt F5=Refresh F9=Retrieve F10=Restart F12=Cancel
F19=Extended system status F24=More keys

Process status (ps)

The i5/OS QShell version of the ps utility displays temporary storage information for each process displayed using the **-o tmpsz** parameter.

For example, to view all the temporary storage in megabytes of all processes executed by the user qsecofr:

```
> ps -u qsecofr -o tmsz,pid,jobid
  TMSZ      PID JOBID
    0       84 093717/ qsecofr /qpadev0002
    9       89 093722/ qsecofr /qpadev0002
    3      293 093962/ qsecofr /qpadev0003
    4      294 093963/ qsecofr /qzshsh
    4      298 093967/ qsecofr /qp0zspwp
$
```

Analyzing I/O problems

This section describes tools that can be used to analyze I/O problems.

Work with System Status (WRKSYSSTS)

You can use this tool to display pools with high paging rates.

In addition to the WRKSYSSTS usage mentioned in “Analyzing memory problems” on page 178, this tool displays pools with high paging rates. Look for pools with high non-database (Non-DB) faulting rates.

Work with Disk Status (WRKDSKSTS)

This tool shows performance and status information for the disk units on the system. In general, disks should have a “% Busy” of less than 40%.

Sample output forWRKDSKSTS

```

                                Work with Disk Status
                                HGWELLS
                                04/28/06 17:01:16

Elapsed time: 00:00:00

Unit  Type      Size      %      I/O      Request  Read  Write  Read  Write  %
      (M)  Used  Rqs  Size (K)  Rqs  Rqs   Rqs   (K)   (K)   Busy
14  6718    13161  34.8    .0      .0      .0    .0    .0    .0    0
15  6718    13161  34.9    .0      .0      .0    .0    .0    .0    0
16  6718    15355  .0      .0      .0      .0    .0    .0    .0    0
17  6718    15355  .0      .0      .0      .0    .0    .0    .0    0
18  6718    15355  .0      .0      .0      .0    .0    .0    .0    0
19  6718    13161  .0      .0      .0      .0    .0    .0    .0    0
20  6718    13161  .0      .0      .0      .0    .0    .0    .0    0
21  6718    15355  .0      .0      .0      .0    .0    .0    .0    0
22  6718    15355  .0      .0      .0      .0    .0    .0    .0    0
23  6718    13161  .0      .0      .0      .0    .0    .0    .0    0
24  6718    15355  .0      .0      .0      .0    .0    .0    .0    0
25  6718    15355  .0      .0      .0      .0    .0    .0    .0    0
26  6718    13161  .0      .0      .0      .0    .0    .0    .0    0
More...

Command
====>
F3=Exit  F5=Refresh  F12=Cancel  F24=More keys
```

Pressing F11 in the WRKDSKSTS display shows additional columns, including the status of each disk unit.

Failed or degraded disks can significantly affect performance in addition to the risk of losing data.

Work with TCP/IP Network Status (WRKTCPSTS or NETSTAT)

This tool shows information about the status of TCP/IP network routes, interfaces, TCP connections, and UDP ports on your local system. You can also use NETSTAT to end TCP/IP connections and to start or stop TCP/IP interfaces.

Sample output for WRKTCPSTS OPTION(*CNN):

```

                                Work with TCP/IP Connection Status
                                System:  HGWELLS

Type options, press Enter.
  3=Enable debug  4=End  5=Display details  6=Disable debug
  8=Display jobs

  Remote      Remote      Local
Opt Address    Port      Port      Idle Time  State
  *           *           ftp-con > 040:31:26 Listen
  *           *           telnet    000:14:12 Listen
  *           *           smtp      076:24:48 Listen
  *           *           netbios > 076:24:50 Listen
  *           *           netbios > 000:01:20 *UDP
  *           *           netbios > 000:01:16 *UDP
  *           *           netbios > 000:37:15 Listen
  *           *           ldap      076:24:59 Listen
  *           *           cifs      000:37:15 Listen
  *           *           drda      074:03:31 Listen
  *           *           ddm       076:25:27 Listen
  *           *           ddm-ssl   076:25:27 Listen

                                More...
F3=Exit  F5=Refresh  F9=Command line  F11=Display byte counts  F12=Cancel
F20=Work with IPv6 connections  F22=Display entire field  F24=More keys

```

Communications trace

The Communications Trace Analyzer tool helps you analyze an iSeries communications trace (taken using the STRCMNTRC or TRCCNN command) for various performance, connection, or security problems you might be experiencing.

To start a communications trace, follow these steps:

1. (Optional) To collect very large traces, set the value for maximum storage size on the system. This value represents the amount of storage, in megabytes, that the communications trace function can allocate to contain the trace data from all traces run. To specify the value for maximum storage size, follow these steps:
 - a. At the command line, type STRSST (Start System Service Tools).
 - b. Type your Service Tools userid and password.
 - c. Select option 1 (Start a Service Tool).
 - d. Select option 3 (Work with communications trace).
 - e. Press F10 (Change size).
 - f. For the New maximum storage size prompt, specify a sufficient amount of storage for the traces you collect, and press Enter.
 - g. Press F3 (Exit) to exit System Service Tools.
2. At the command line, type STRCMNTRC.
3. At the Configuration object prompt, specify the name of the line, such as TRNLINE.
4. At the Type prompt, specify the type of resource, either *LIN or *NWI.
5. At the Buffer size prompt, specify a sufficient amount of storage for the anticipated volume of data. For most protocols, 8 MB is sufficient storage. For a 10/100 Ethernet connection, 16 MB to 1 GB is sufficient. If you are uncertain, specify 16 MB for the maximum amount of storage allowed for the protocol.
6. At the Communications trace options prompt, specify *RMTIPADR if you want to limit the data collected to a trace of one remote interface. Otherwise, use the default value.

- At the Remote IP address prompt, specify the IP address associated with the remote interface to which the trace data will be collected.

The communications trace runs until one of the following situations occur:

- The ENDCMNTRC command is run.
- A physical line problem causes the trace to end.
- The Trace full prompt specifies *STOPTRC and the buffer becomes full.

You can print the communications trace data from two different sources, depending on how you collected the trace. For IPv4, you can print from the raw data you collected, or you can print from a stream file in which you previously dumped the raw data. For IPv6, you must print from a stream file.

Sample communications trace (partial):

```

COMMUNICATIONS TRACE                               Title:                                04/20/06 15:57:51                               Page:
Record   Data   Record   Controller   Destination   Source   Frame   Number   Number   Poll/
Number   S/R   Length   Timer        MAC Address   MAC Address   Format   Sent     Received   Final
-----
1    R    46    15:56:49.333386    00112508544B  40007F3704A2  ETHV2   Type: 0800
Frame Type : IP          DSCP: 26  ECN: 00-NECT Length: 40   Protocol: TCP          Datagram :
Src Addr: 9.10.72.171    Dest Addr: 9.5.8.62      Fragment Flags: DON'T,
IP Header : 456800282E8440007A066EEC090A48AB0905083E
IP Options : NONE
TCP . . . : Src Port: 3697,Unassigned  Dest Port: 23,TELNET
SEQ Number: 1115323641 ('427A7CF9'X) ACK Number: 1173496341 ('45F22215'X)
Code Bits: ACK          Window: 63522 TCP Option: NONE
TCP Header : 0E710017427A7CF945F222155010F8221EB70000
Data . . . . : 0000000000000000 *.....
2    R    46    15:56:49.381549    FFFFFFFF      40007F3704A2  ETHV2   Type: 0806
Frame Type : ARP          Src Addr: 9.5.64.4      Dest Addr: 9.5.64.171   Operation: REQU
ARP Header : 000108000604000140007F3704A2090540040000000000090540AB
Data . . . . : 0000000000000000 0000000000000000 0000 *.....
3    R    46    15:56:49.382557    FFFFFFFF      0006296B427D  ETHV2   Type: 0806
Frame Type : ARP          Src Addr: 9.5.149.243   Dest Addr: 9.5.149.129   Operation: REQU
ARP Header : 00010800060400010006296B427D090595F3FFFFFFFFFFFF09059581
Data . . . . : 0000000000000000 0000000000000000 0000 *.....
4    R    46    15:56:49.382603    FFFFFFFF      0006296B427D  ETHV2   Type: 0806
Frame Type : ARP          Src Addr: 9.5.149.243   Dest Addr: 9.5.149.129   Operation: REQU
ARP Header : 00010800060400010006296B427D090595F3FFFFFFFFFFFF09059581
Data . . . . : 0000000000000000 0000000000000000 0000 *.....
5    R    46    15:56:49.382732    FFFFFFFF      0006296B427D  ETHV2   Type: 0806
F3=Exit  F12=Cancel  F19=Left  F20=Right  F24=More keys

```

Print from raw data collected:

If you collected the raw data without dumping it, follow these steps to print the data.

The steps are:

- At the command line, type PRTCMNTRC.
- At the Configuration object prompt, specify the same line you specified when you started the trace, such as TRNLINE, and press Enter.
- At the Type prompt, specify the type of resource, either *LIN or *NWI.
- At the Character code prompt, specify either *EBCDIC or *ASCII. You should print the data twice, once specifying *EBCDIC and once specifying *ASCII.
- At the Format TCP/IP data prompt, type *YES, and press Enter twice.
- Perform steps 1 through 5 again, specifying the other character code.

Print from stream file:

If you dumped the data to a stream file, follow these steps to print the data.

The steps are:

1. At the command line, type `PRTCMNTRC`.
2. At the From stream file prompt, specify the path name, such as `/mydir/mytraces/trace1`, and press Enter.
3. At the Character code prompt, specify `*EBCDIC` or `*ASCII`. You should print the data twice, once specifying `*EBCDIC` and once specifying `*ASCII`.
4. Perform steps 1 through 3 again, specifying the other character code.

For more information, see <http://publib.boulder.ibm.com/iseres/>.

Diagnosing crashes

You can try a number of approaches when determining the cause of a crash. The process normally involves isolating the problem by checking the system setup and trying various diagnostic options.

Checking the system environment

The system might have been in a state that has caused the VM to crash. For example, there could be a resource shortage (such as memory or disk) or a stability problem.

Check the `javacore` file, which contains system information (see Chapter 22, “Using Javadump,” on page 245). The `javacore` file tells you how to find disk and memory resource information. The system logs can give indications of system problems.

Finding out about the Java environment

Use the `javacore` file to determine what each thread was doing and which Java methods were being executed. Use the `-verbose:gc` option to look at the state of the Java heap.

Use the `-verbose:gc` option to determine if:

- A shortage of Java heap space could have caused the crash.
- The crash occurred during garbage collection, indicating a possible garbage collection fault. See Chapter 2, “Memory management,” on page 7.

Detailed crash diagnosis

If the analysis described in previous sections is not sufficient to determine the cause of the crash, you might have to analyze the system core file in more detail.

For details, see “Using dbx” on page 186.

Diagnosing hangs

The VM is hanging if the process is still present but is not responding.

A hang can occur if:

- 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.

i5/OS deadlocks

If two or more Java threads should be executing work but are idle, the cause might be a deadlock.

For an explanation of deadlocks and how the Javdump tool is used to diagnose them, see “Locks, monitors, and deadlocks (LOCKS)” on page 252. Some threads in the job might be busy and consuming CPU even if a deadlock is preventing other threads from running.

i5/OS busy hangs

If no threads are deadlocked there are other possible reasons why threads might be idle.

Threads might be idle because:

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 (for example, sysRecv) waiting to do work.

The first two reasons imply a fault in the Java code, either in the application itself or in libraries on which the application depends (including, in rare cases, the standard class libraries included in the SDK).

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

If the process is using processor cycles, either it has entered an infinite loop or it is suffering from very bad performance. Using WRKACTJOB, you can work with individual threads.

After entering WRKACTJOB, locate your Java VM in the list of jobs, and put a "5-Work with" next to the VM in the Opt Column. On the next screen, use option "20-Work with threads, if active". You can now determine which threads are using the CPU time. If the process has entered an infinite loop, a small number of threads are probably using all the time. Viewing the stack for these threads can provide clues about the source of the problem.

Understanding memory usage

Because IBM Technology for Java runs in the i5/OS Portable Application Solutions Environment (i5/OS PASE) in a 32-bit environment, it uses a different memory model from the one used by most i5/OS applications. i5/OS PASE and IBM Technology for Java manage this memory and its interactions with the rest of the system automatically. However, some understanding of the memory model can be helpful when debugging, especially when working with native code.

The 32-bit i5/OS PASE Virtual memory model

i5/OS PASE assigns a 32-bit virtual address space partitioned into 16 segments of 256 MB each. Process addressability to data is managed at the segment level, so a data segment can either be shared (between processes) or private.

Use the MAXDATA setting to control the memory model. By default, the IBM Technology for Java launcher alters its MAXDATA setting in response to the command-line options to optimize the amount of memory available to the process. The defaults are as follows:

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

Override these values by setting the environment variable **LDR_CNTRL=MAXDATA=<value>**. See “Changing the Memory Model (32-bit JVM)” on page 107 for the possible values and an explanation of their meanings. The defaults are appropriate for most applications, therefore setting this value explicitly is rarely necessary.

The process and garbage-collected heaps

The VM maintains two memory areas: the garbage-collected Java heap, and the process (or native) heap. These two heaps have different purposes, are maintained by different mechanisms, and are largely independent of each other.

The garbage-collected heap contains instances of Java objects and is often referred to as “the heap”. This heap is managed by the garbage collector. Use the heap (for example: **-Xms** and **-Xmx**) command-line settings to configure this heap. Internally, the garbage-collected heap is allocated using mmap or shmat. The maximum size of this heap is preallocated during VM 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 using heap expansion. See Chapter 2, “Memory management,” on page 7 for more information.

The JVM allocates the process heap using the underlying malloc and free mechanisms of the operating system. The process heap is used for the underlying implementation of particular Java objects; for example, malloc allocations by application JNI code, compiled code generated by the Just In Time (JIT) Compiler, and threads to map to Java threads.

Monitoring the garbage-collected heap

The most straightforward, and often most useful, way to monitor the garbage-collected heap is by monitoring garbage collection. Use the **-verbose:gc** option to enable a report on stderr each time garbage collection occurs. You can also direct this output to a log file using the **-Xverbosegclog:<filename>** option.

See Chapter 28, “Garbage Collector diagnostics,” on page 329 for more information on verbosegc output and monitoring.

Process heap usage

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

You cannot directly set the size of the process heap. Instead, the process heap uses memory in the 32-bit address space that is not used by the garbage-collected heap. To increase the size of the process heap, decrease the maximum Java heap size (**-Xmx** option).

The process heap will typically grow to a stable size and then stay at a similar size. One exception is the compilation of JIT code. Space for the compiled code is allocated from the process heap using `malloc()`. This compilation can cause a slow increase in process heap usage as little-used methods reach the threshold to undergo JIT compilation.

You can monitor the JIT compilation of code to avoid confusing this behavior with a memory leak using the command-line option **"-Xjit:verbose={compileStart|compileEnd}"**. (Note that this option must be surrounded with quotation marks so that the vertical bar is not interpreted by the shell.)

OutOfMemoryError exceptions

An `OutOfMemoryError` exception occurs when either the garbage-collected heap or the process heap has run out of space. A heap can run out of space because of a lack of memory available in the heap or because of a memory leak.

If the problem occurs because of a 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 provide a temporary solution for a production system. Solving the problem requires finding the source of the leak.

Some `OutOfMemoryError` exceptions also carry an explanatory message, including an error code. See Appendix C, "Messages," on page 419 for more information on any messages received.

Most `OutOfMemoryError` exceptions are caused by exhausting the garbage-collected heap. Therefore, if no error message is present, the first stage is to monitor the garbage-collected heap using **-verbose:gc**. If this heap does not seem to be exhausted, the problem might be with the process heap.

Garbage-collected heap exhaustion

The garbage-collected 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 the thread stacks (see Chapter 2, "Memory management," on page 7).

You can identify garbage-collected heap exhaustion using the **-verbose:gc** output. When the heap is becoming exhausted, garbage collection occurs more and more frequently, with less memory being freed. Eventually the VM will fail, and the heap occupancy will be at, or almost at, 100%.

If the garbage-collected 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. Look for suspect data structures that are referencing large numbers of Java objects that should have been released. See Chapter 23, "Using Heapdump," on page 257. You can obtain similar information by using third-party tools.

Submitting a bug report

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

The process for raising a bug is detailed in Part 2, “Submitting problem reports,” on page 81. The following data should be included in the bug report:

Required:

1. The OutOfMemory condition. The error with any message or stack trace that accompanied it.
2. **-verbose:gc** output. (Even if the problem is determined to be native heap exhaustion)

As appropriate:

1. The Heapdump output
2. The javacore.txt file

Using dbx

dbx is the AIX standard command-line debugger. As on AIX, using dbx on i5/OS to debug or inspect a process not owned by the dbx user requires special levels of system-wide authority. This authority is like the “root” authority in UNIX.

See the PASE documentation for details on using dbx.

Interactive use of dbx on i5/OS must be from a shell that has been properly configured. Other interactive command-line tools such as jextract and jconsole also require a properly configured shell. See “Diagnosing problems at the command line” on page 173 for more information.

You can automate dbx into diagnostic “probes”. IBM support might ask you to obtain and run selected probes. The probe might be against a test instance of the troubled application, or against the dump files generated by an application failure.

Probes might be just a few lines of code, and thus easy to provide as attachments or inline in e-mail text. You might also be asked to use the `/fromibm/os400` directory on the IBM anonymous FTP public server, named `ftp.emea.ibm.com`. In general, careful diagnostic probing provides IBM with local diagnostic information for your problem, while minimizing the data transfer required to arrive at a solution.

Using the DBX Plug-in for Java

The DBX Plug-in for Java is designed to be used with the UNIX standard debugger dbx (available on i5/OS PASE) to provide additional Java-specific capabilities. Although the DBX Plug-in is supplied as part of the SDK, it is not supported. However, IBM will accept bug reports.

There are two known bugs in the i5/OS support:

1. There is a known problem attaching to different threads after the DBX Plug-in is loaded. When you have loaded the DBX Plug-in, do not run a ‘thread’ or ‘thread info’ command at the (dbx) prompt, as this will abruptly end dbx. If you need to use the dbx ‘thread’ command, use it before loading the DBX Plug-in.
2. An application might not continue execution after stopping at a dbx-specified breakpoint. This level of debugging, including breakpoint use and live debugging, is beyond the scope of this document.

To use the DBX Plug-in, you need a version of dbx that supports the pluginload command. (The version of dbx shipped with i5/OS PASE supports this feature.)

Start dbx and enter the pluginload command to load the DBX Plug-in.

```
(dbx) pluginload libdbx_j9.so
```

If the Plug-in does not load correctly:

- You might not have the necessary authorities on the system
- The core file might be incomplete or invalid.

For a brief summary of the commands available in the DBX Plug-in, type:

```
(dbx) j9help
```

Example: testing the integrity of the heap image

This example tests the integrity of a heap image in a core file.

The example has the following steps:

1. Start dbx using your core file:

```
$ dbx -W core.20060421.063015.4253.dmp
```

2. Load the DBX Plug-in for Java:

```
(dbx) pluginload libdbx_j9.so
```

3. Check the integrity of the heap:

```
(dbx) plugin j9 gccheck
```

This command finds the VM in the core file and uses the GC component to validate the heap contents.

4. Unload the DBX Plug-in for Java:

```
(dbx) pluginunload j9
```

Important dbx usage notes and warnings

This guide is not intended as a comprehensive guide to using dbx; this guide is intended as platform-specific information to be used by developers and service personnel familiar with dbx.

The AIX documentation for dbx contains more general information. See http://publib.boulder.ibm.com/infocenter/pseries/v5r3/topic/com.ibm.aix.doc/aixprgpd/genprogc/dbx_symbolic_debug.htm.

Note: Do not attempt to use dbx, either interactively or as part of a diagnostic probe, to attach to any active business-critical applications, unless specifically directed to do so by IBM service personnel.

1. When you use dbx to attach to a running process (with or without the DBX Plug-in), it causes that process to stop. If you use the quit command to exit dbx, any process to which dbx is currently attached will also be terminated. To detach dbx from a process, without terminating the process, you must use the detach command.
2. Use of dbx and the DBX Plug-in to diagnose a running process involves the risk of terminating the target process abruptly or of causing the target application to exhibit timing-related failures. Any diagnostic dbx-based scripts provided by IBM support are designed to minimize their potential affect, but the possibility of disturbing a running application is unavoidable.

Using dbx to investigate a Java system dump

When an AIX application running in i5/OS PASE ends abruptly, it might attempt to write a platform-specific core file to the file system. The format of this file is defined by the C runtime library and the file is automatically created if necessary.

See the documentation for `abort()` in the C runtime library and “Enabling i5/OS PASE core files” on page 169.

If the virtual machine for Java is also running in that failing process, the VM might attempt to write additional diagnostic output files. These files are described in “Summary of diagnostic information” on page 217. These files are packaged for submission to IBM by the `jextract` command. See Part 2, “Submitting problem reports,” on page 81 for more information about data to collect.

A core file without additional Java-related is called:

`core`

A VM system dump file providing additional information accessible using the DBX Plug-in for Java is called:

`core.{date}.{time}.{pid}.dmp`

The VM system dump file is the same file specified to the `jextract` command.

If a Java failure results in a simple core file, but the VM cannot successfully create the VM system dump, `dbx` might still be useful. However, the added function of the DBX Plug-in for Java is not available.

Starting dbx on a system dump (core.{date}.{time}.{pid}.dmp)

To start `dbx` on a system dump use the command `dbx -W <filename>`.

After ensuring your environment is correctly configured (see “Diagnosing problems at the command line” on page 173), use the command `dbx -W <filename>`. For example, if you have a VM system dump file named `core.20060421.063015.4253.dmp`, enter the command:

```
$ dbx -W core.20060421.063015.4253.dmp
```

The use of the `-W` flag tells `dbx` to determine the name of the top-level application program directly from the “map” in the core file itself.

Chapter 15. Sun Solaris problem determination

IBM does not supply a software developer kit or runtime environment for the Sun Solaris platform. However, IBM does make strategic products, such as the WebSphere Application Server, for this platform. In this case, the WebSphere Application Server contains an embedded copy of the Sun Solaris JVM alongside IBM enhancements, including all the security, ORB, and XML technologies provided on other platforms by IBM. The WebSphere Application Server Solaris SDK is therefore a hybrid of Sun and IBM products but the core JVM and JIT are Sun Solaris.

This *Information Center* is therefore not appropriate for diagnosis on Sun Solaris. IBM does service the Sun Solaris SDK, but only when it is an embedded part of IBM middleware, for example, WebSphere Application Server. If you get a Java problem on Solaris *as a result of using an IBM middleware product*, go to Part 2, “Submitting problem reports,” on page 81 and submit a bug report.

For problems on the Sun Solaris platform, you are advised to look at:
http://java.sun.com/j2se/1.5/pdf/jdk50_ts_guide.pdf.

Chapter 16. Hewlett-Packard SDK problem determination

IBM does not supply a software developer kit or runtime environment for HP platforms. However, IBM does make strategic products, such as the WebSphere Application Server, for this platform. In this case, the WebSphere Application Server contains an embedded copy of the HP JVM alongside IBM enhancements, including all the security, ORB, and XML technologies provided on other platforms by IBM. The WebSphere Application Server HP SDK is therefore a hybrid of HP and IBM products but the core JVM and JIT are HP software.

This *Information Center* is therefore not appropriate for diagnosis on HP platforms. IBM does service the HP SDK, but only when it is an embedded part of IBM middleware, for example, WebSphere Application Server. If you get a Java problem on an HP platform *as a result of using an IBM middleware product*, go to Part 2, “Submitting problem reports,” on page 81 and submit a bug report.

For problems on HP platforms, you are advised to look at: http://h18012.www1.hp.com/java/support/troubleshooting_guide.html.

Chapter 17. ORB problem determination

One of your first tasks when debugging an ORB problem is to determine whether the problem is in the client-side or in the server-side of the distributed application. Think of a typical RMI-IIOP session as a simple, synchronous communication between a client that is requesting access to an object, and a server that is providing it.

During this communication, a problem might occur in the execution of one of the following steps:

1. The client writes and sends a request to the server.
2. The server receives and reads the request.
3. The server executes the task in the request.
4. The server writes and sends a reply back.
5. The client receives and reads the reply.

It is not always easy to identify where the problem occurred. Often, the information that the application returns, in the form of stack traces or error messages, is not enough for you to make a decision. Also, because the client and server communicate through their ORBs, if a problem occurs, both sides will probably record an exception or unusual behavior.

This section describes all the clues that you can use to find the source of the ORB problem. It also describes a few common problems that occur more frequently. The topics are:

- “Identifying an ORB problem”
- “Debug properties” on page 194
- “ORB exceptions” on page 195
- “Interpreting the stack trace” on page 198
- “Interpreting ORB traces” on page 199
- “Common problems” on page 202
- “IBM ORB service: collecting data” on page 205

Identifying an ORB problem

A background of the constituents of the IBM ORB component.

What the ORB component contains

The ORB component contains the following:

- Java ORB from IBM and rmi-iiop runtime (com.ibm.rmi.*, com.ibm.CORBA.*)
- RMI-IIOP API (javax.rmi.CORBA.*, org.omg.CORBA.*)
- IDL to Java implementation (org.omg.* and IBM versions com.ibm.org.omg.*)
- Transient name server (com.ibm.CosNaming.*, org.omg.CosNaming.*) - tnameserv
- -iiop and -idl generators (com.ibm.tools.rmi.rmic.*) for the rmic compiler - rmic
- idlj compiler (com.ibm.idl.*)

What the ORB component does not contain

The ORB component does *not* contain:

- RMI-JRMP (also known as Standard RMI)
- JNDI and its plug-ins

Therefore, if the problem is in `java.rmi.*` or `sun.rmi.*`, it is not an ORB problem. Similarly, if the problem is in `com.sun.jndi.*`, it is not an ORB problem.

Platform dependent problems

If possible, run the test case on more than one platform. All the ORB code is shared. You can nearly always reproduce genuine ORB problems on any platform. If you have a platform-specific problem, it is likely to be in some other component.

JIT problem

JIT bugs are very difficult to find. They might show themselves as ORB problems. When you are debugging or testing an ORB application, it is always safer to switch off the JIT by setting the option **-Xint**.

Fragmentation

Disable fragmentation when you are debugging the ORB. Although fragmentation does not add complications to the ORB's functioning, a fragmentation bug can be difficult to detect because it will most likely show as a general marshalling problem. The way to disable fragmentation is to set the ORB property **com.ibm.CORBA.FragmentSize=0**. You must do this on the client side and on the server side.

ORB versions

The ORB component carries a few version properties that you can display by calling the main method of the following classes:

1. `com.ibm.CORBA.iiop.Version` (ORB runtime version)
2. `com.ibm.tools.rmic.iiop.Version` (for tools; for example, `idlj` and `rmic`)
3. `rmic -iiop -version` (run the command line for `rmic`)

Limitation with bidirectional GIOP

Bidirectional GIOP is not supported.

Debug properties

Properties to use to enable ORB traces.

Attention: Do not enable tracing for normal operation, because it might cause performance degradation. Even if you have switched off tracing, FFDC (First Failure Data Capture) is still working, so that only serious errors are reported. If a debug file is produced, examine it to check on the problem. For example, the server might have stopped without performing an `ORB.shutdown()`.

You can use the following properties to enable the ORB traces:

- **com.ibm.CORBA.Debug:** This property turns on trace, message, or both. If you set this property to **trace**, only traces are enabled; if set to **message**, only messages are enabled. When set to **true**, both types are enabled; when set to **false**, both types are disabled. The default is **false**.
- **com.ibm.CORBA.Debug.Output:** This property redirects traces to a file, which is known as a trace log. When this property is not specified, or it is set to an empty string, the file name defaults to the format `orbtrc.DDMMYYYY.HHmm.SS.txt`, where D=Day; M=Month; Y=Year; H=Hour (24 hour format); m=Minutes; S=Seconds. If the application (or Applet) does not have the privilege that it requires to write to a file, the trace entries go to `stderr`.
- **com.ibm.CORBA.CommTrace:** This property turns on wire tracing, also known as Comm tracing. Every incoming and outgoing GIOP message is sent to the trace log. You can set this property independently from Debug. This property is useful if you want to look only at the flow of information, and you are not interested in debugging the internals. The only two values that this property can have are **true** and **false**. The default is **false**.

Here is an example of common usage:

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

For `rmic -iiop` or `rmic -idl`, the following diagnostic tools are available:

- **-J-Djavac.dump.stack=1:** This tool ensures that all exceptions are caught.
- **-Xtrace:** This tool traces the progress of the parse step.

If you are working with an IBM SDK, you can obtain `CommTrace` for the transient name server (`tnameserv`) by using the standard environment variable **IBM_JAVA_OPTIONS**. In a separate command session to the server or client SDKs, you can use:

```
set IBM_JAVA_OPTIONS=-Dcom.ibm.CORBA.CommTrace=true -Dcom.ibm.CORBA.Debug=true
```

or the equivalent platform-specific command.

The setting of this environment variable affects each Java process that is started, so use this variable carefully. Alternatively, you can use the **-J** option to pass the properties through the `tnameserv` wrapper, as follows:

```
tnameserv -J-Dcom.ibm.CORBA.Debug=true
```

ORB exceptions

The exceptions that can be thrown are split into user and system categories.

If your problem is related to the ORB, unless your application is doing nothing or giving you the wrong result, your log file or terminal is probably full of exceptions that include the words “CORBA” and “rmi” many times. All unusual behavior that occurs in a good application is highlighted by an exception. This principle also applies for the ORB with its CORBA exceptions. Similarly to Java, CORBA divides its exceptions into user exceptions and system exceptions.

User exceptions

User exceptions are IDL defined and inherit from `org.omg.CORBA.UserException`. These exceptions are mapped to checked exceptions in Java; that is, if a remote method raises one of them, the application that called that method must catch the exception. User exceptions are usually not fatal exceptions and should always be

handled by the application. Therefore, if you get one of these user exceptions, you know where the problem is, because the application developer had to make allowance for such an exception to occur. In most of these cases, the ORB is not the source of the problem.

System exceptions

System exceptions are thrown transparently to the application and represent an unusual condition in which the ORB cannot recover gracefully, such as when a connection is dropped. The CORBA 2.6 specification defines 31 system exceptions and their mapping to Java. They all belong to the `org.omg.CORBA` package. The CORBA specification defines the meaning of these exceptions and describes the conditions in which they are thrown.

The most common system exceptions are:

- **BAD_OPERATION:** This exception is thrown when an object reference denotes an existing object, but the object does not support the operation that was called.
- **BAD_PARAM:** This exception is thrown when a parameter that is passed to a call is out of range or otherwise considered not valid. An ORB might raise this exception if null values or null pointers are passed to an operation.
- **COMM_FAILURE:** This exception is raised if communication is lost while an operation is in progress, after the request was sent by the client, but before the reply from the server has been returned to the client.
- **DATA_CONVERSION:** This exception is raised if an ORB cannot convert the marshaled representation of data into its native representation, or cannot convert the native representation of data into its marshaled representation. For example, this exception can be raised if wide character codeset conversion fails, or if an ORB cannot convert floating point values between different representations.
- **MARSHAL:** This exception indicates that the request or reply from the network is structurally not valid. This error typically indicates a bug in either the client-side or server-side runtime. For example, if a reply from the server indicates that the message contains 1000 bytes, but the actual message is shorter or longer than 1000 bytes, the ORB raises this exception.
- **NO_IMPLEMENT:** This exception indicates that although the operation that was called exists (it has an IDL definition), no implementation exists for that operation.
- **UNKNOWN:** This exception is raised if an implementation throws a non-CORBA exception, such as an exception that is specific to the implementation's programming language. It is also raised if the server returns a system exception that is unknown to the client. If the server uses a later version of CORBA than the version that the client is using, and new system exceptions have been added to the later version this exception can happen.

Completion status and minor codes

Two pieces of data are associated with each system exception, these are described in this section.

- A completion status, which is an enumerated type that has three values: `COMPLETED_YES`, `COMPLETED_NO` and `COMPLETED_MAYBE`. These values indicate either that the operation was executed in full, that the operation was not executed, or that the execution state cannot be determined.
- A long integer, called minor code, that can be set to some ORB vendor-specific value. CORBA also specifies the value of many minor codes.

Usually the completion status is not very useful. However, the minor code can be essential when the stack trace is missing. In many cases, the minor code identifies the exact location of the ORB code where the exception is thrown and can be used by the vendor's service team to localize the problem quickly. However, for standard CORBA minor codes, this is not always possible. For example:

```
org.omg.CORBA.OBJECT_NOT_EXIST: SERVANT_NOT_FOUND    minor code: 4942FC11    completed: No
```

Minor codes are usually expressed in hexadecimal notation (except for Sun's minor codes, which are in decimal notation) that represents four bytes. The OMG organization has assigned to each vendor a range of 4096 minor codes. The IBM vendor-specific minor code range is 0x4942F000 through 0x4942FFFF. Appendix A, "CORBA minor codes," on page 409 gives diagnostic information for common minor codes.

System exceptions might also contain a string that describes the exception and other useful information. You will see this string when you interpret the stack trace.

The ORB tends to map all Java exceptions to CORBA exceptions. A runtime exception is mapped to a CORBA system exception, while a checked exception is mapped to a CORBA user exception.

More exceptions other than the CORBA exceptions could be generated by the ORB component in a code bug. All the Java unchecked exceptions and errors and others that are related to the ORB tools `rmic` and `idlj` must be considered. In this case, the only way to determine whether the problem is in the ORB, is to look at the generated stack trace and see whether the objects involved belong to ORB packages.

Java security permissions for the ORB

When running with a Java SecurityManager, invocation of some methods in the CORBA API classes might cause permission checks to be made that could result in a `SecurityException`.

The following table shows methods affected when running with Java 2 SecurityManager:

Class/Interface	Method	Required permission
org.omg.CORBA.ORB	init	java.net.SocketPermission resolve
org.omg.CORBA.ORB	connect	java.net.SocketPermission listen
org.omg.CORBA.ORB	resolve_initial_references	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_is_a	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_non_existent	java.net.SocketPermission connect

Class/Interface	Method	Required permission
org.omg.CORBA. portable.ObjectImpl	OutputStream _request (String, boolean)	java.net.SocketPermission connect
org.omg.CORBA. portable.ObjectImpl	_get_interface_def	java.net.SocketPermission connect
org.omg.CORBA. Request	invoke	java.net.SocketPermission connect
org.omg.CORBA. Request	send_deferred	java.net.SocketPermission connect
org.omg.CORBA. Request	send_oneway	java.net.SocketPermission connect
javax.rmi. PortableRemoteObject	narrow	java.net.SocketPermission connect

If your program uses any of these methods, ensure that it is granted the necessary permissions.

Interpreting the stack trace

Whether the ORB is part of a middleware application or you are using a Java stand-alone application (or even an applet), you must retrieve the stack trace that is generated at the moment of failure. It could be in a log file, or in your terminal or browser window, and it could consist of several chunks of stack traces.

The following example describes a stack trace that was generated by a server ORB running in the WebSphere Application Server:

```
org.omg.CORBA.MARSHAL: com.ibm.ws.pmi.server.DataDescriptor; IllegalAccessException minor code: 4942F23E completed: No
at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:199)
at com.ibm.rmi.ioop.CDRInputStream.read_value(CDRInputStream.java:1429)
at com.ibm.rmi.io.ValueHandlerImpl.read_Array(ValueHandlerImpl.java:625)
at com.ibm.rmi.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:273)
at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:189)
at com.ibm.rmi.ioop.CDRInputStream.read_value(CDRInputStream.java:1429)
at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie._invoke(_EJSRemoteStatelessPmiService_Tie.java:613)
at com.ibm.CORBA.ioop.ExtendedServerDelegate.dispatch(ExtendedServerDelegate.java:515)
at com.ibm.CORBA.ioop.ORB.process(ORB.java:2377)
at com.ibm.CORBA.ioop.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 `IllegalAccessExpection` occurred that was associated to class `com.ibm.ws.pmi.server.DataDescriptor`. This information is an indication of the real problem and should be investigated first.

Interpreting ORB traces

The ORB trace file contains messages, trace points, and wire tracing. This section describes the various types of trace.

Message trace

An example of a message trace.

Here is a simple example of a message:

```
19:12:36.306 com.ibm.rmi.util.Version logVersions:110 P=754534:0=0:CT
ORBRas[default] IBM Java ORB build orbdev-20050927
```

This message records the time, the package, and the method name that was called. In this case, `logVersions()` prints out, to the log file, the version of the running ORB.

After the first colon in the example message, the line number in the source code where that method invocation is done is written (110 in this case). Next follows the letter P that is associated with the process number that was running at that moment. This number is related (by a hash) to the time at which the ORB class was loaded in that process. It is unlikely that two different processes load their ORBs at the same time.

The following `O=0` (alphabetic O = numeric 0) indicates that the current instance of the ORB is the first one (number 0). CT specifies that this is the main (control) thread. Other values are: LT for listener thread, RT for reader thread, and WT for worker thread.

The `ORBRas` field shows which RAS implementation the ORB is running. It is possible that when the ORB runs inside another application (such as a WebSphere application), the ORB RAS default code is replaced by an external implementation.

The remaining information is specific to the method that has been logged while executing. In this case, the method is a utility method that logs the version of the ORB.

This example of a possible message shows the logging of entry or exit point of methods, such as:

```
14:54:14.848 com.ibm.rmi.iiop.Connection <init>:504 LT=0:P=650241:0=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:

```
// 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=852270:0=0:CT
Local Port:    4899 (0x1323)
Local IP:      9.20.178.136
Remote Port:   4893 (0x131D)
Remote IP:     9.20.178.136
GIOP Version:  1.2
Byte order:    big endian

Fragment to follow: No // This is the last fragment of the request
Message size: 276 (0x114)
--

Request ID:      5 // Request Ids are in ascending sequence
Response Flag:   WITH_TARGET // it means we are expecting a reply to this request
Target Address:  0
Object Key:      length = 26 (0x1A) // the object key is created by the server when exporting
                                   // the servant and retrieved in the IOR using a naming service
                4C4D4249 00000010 14F94CA4 00100000
                00080000 00000000 0000
Operation:       message // That is the name of the method that the client invokes on the servant
Service Context: length = 3 (0x3) // There are three service contexts
Context ID:      1229081874 (0x49424D12) // Partner version service context. IBM only
Context data:    length = 8 (0x8)
                00000000 14000005

Context ID:      1 (0x1) // Codeset CORBA service context
Context data:    length = 12 (0xC)
                00000000 00010001 00010100

Context ID:      6 (0x6) // Codebase CORBA service context
Context data:    length = 168 (0xA8)
                00000000 00000028 49444C3A 6F6D672E
                6F72672F 53656E64 696E6743 6F6E7465
                78742F43 6F646542 6173653A 312E3000
                00000001 00000000 00000006 00010200
                0000000D 392E3230 2E313738 2E313336
                00001324 0000001A 4C4D4249 00000010
                15074A96 00100000 00080000 00000000
                00000000 00000002 00000001 00000018
```



```

00000000 00010001 00000001 00010020
00010100 00000000 49424D0A 00000008
00000000 14000005
Data Offset: 11c
// raw data that goes in the wire in numbered rows of 16 bytes and the corresponding ASCII
decoding
0000: 47494F50 01020000 00000114 00000005  GIOP.....
0010: 03000000 00000000 0000001A 4C4D4249  ....LMBI
0020: 00000010 14F94CA4 00100000 00080000  ....L.....
0030: 00000000 00000000 00000008 6D657373  ....mess
0040: 61676500 00000003 49424D12 00000008  age....IBM....
0050: 00000000 14000005 00000001 0000000C  ....
0060: 00000000 00010001 00010100 00000006  ....
0070: 000000A8 00000000 00000028 49444C3A  ....(IDL:
0080: 6F6D672E 6F72672F 53656E64 696E6743  omg.org/SendingC
0090: 6F6E7465 78742F43 6F646542 6173653A  ontext/CodeBase:
00A0: 312E3000 00000001 00000000 0000006C  1.0.....l
00B0: 00010200 0000000D 392E3230 2E313738  ....9.20.178
00C0: 2E313336 00001324 0000001A 4C4D4249  .136...$.LMBI
00D0: 00000010 15074A96 00100000 00080000  ....J.....
00E0: 00000000 00000000 00000002 00000001  ....
00F0: 00000018 00000000 00010001 00000001  ....
0100: 00010020 00010100 00000000 49424D0A  ...IBM.
0110: 00000008 00000000 14000005 00000000  ....

```

Note: The italic comments that start with a double slash have been added for clarity; they are not part of the traces.

In this example trace, you can see a summary of the principal fields that are contained in the message, followed by the message itself as it goes in the wire. In the summary are several field name-value pairs. Each number is in hexadecimal notation.

For details of the structure of a GIOP message, see the CORBA specification, chapters 13 and 15: <http://www.omg.org/cgi-bin/doc?formal/99-10-07>.

Client or server

From the first line of the summary of the message, you can identify whether the host to which this trace belongs is acting as a server or as a client. OUT GOING means that the message has been generated on the workstation where the trace was taken and is sent to the wire.

In a distributed-object application, a server is defined as the provider of the implementation of the remote object to which the client connects. In this work, however, the convention is that a client sends a request while the server sends back a reply. In this way, the same ORB can be client and server in different moments of the rmi-iiop session.

The trace shows that the message is an outgoing request. Therefore, this trace is a client trace, or at least part of the trace where the application acts as a client.

Time information and host names are reported in the header of the message.

The Request ID and the Operation (“message” in this case) fields can be very helpful when multiple threads and clients destroy the logical sequence of the traces.

The GIOP version field can be checked if different ORBs are deployed. If two different ORBs support different versions of GIOP, the ORB that is using the more

recent version of GIOP should fall back to a common level. By checking that field, however, you can easily check whether the two ORBs speak the same language.

Service contexts

The header also records three service contexts, each consisting of a context ID and context data.

A service context is extra information that is attached to the message for purposes that can be vendor-specific such as the IBM Partner version that is described in the IOR in Chapter 7, “The ORB,” on page 43.

Usually, a security implementation makes extensive use of these service contexts. Information about an access list, an authorization, encrypted IDs, and passwords could travel with the request inside a service context.

Some CORBA-defined service contexts are available. One of these is the Codeset.

In the example, the codeset context has ID 1 and data 00000000 00010001 00010100. Bytes 5 through 8 specify that characters that are used in the message are encoded in ASCII (00010001 is the code for ASCII). Bytes 9 through 12 instead are related to wide characters.

The default codeset is UTF8 as defined in the CORBA specification, although almost all Windows and UNIX platforms typically communicate through ASCII. i5/OS and Mainframes such as zSeries systems are based on the IBM EBCDIC encoding.

The other CORBA service context, which is present in the example, is the Codebase service context. It stores information about how to call back to the client to access resources in the client such as stubs, and class implementations of parameter objects that are serialized with the request.

Common problems

This section describes some of the problems that you might find.

ORB application hangs

One of the worst conditions is when the client, or server, or both, hang. If a hang occurs, the most likely condition (and most difficult to solve) is a deadlock of threads. In this condition, it is important to know whether the workstation on which you are running has more than one CPU, and whether your CPU is using Simultaneous Multithreading (SMT).

A simple test that you can do is to keep only one CPU running, disable SMT, and see whether the problem disappears. If it does, you know that you must have a synchronization problem in the application.

Also, you must understand what the application is doing while it hangs. Is it waiting (low CPU usage), or it is looping forever (almost 100% CPU usage)? Most of the cases are a waiting problem.

You can, however, still identify two cases:

- Typical deadlock
- Standby condition while the application waits for a resource to arrive

An example of a standby condition is where the client sends a request to the server and stops while waiting for the reply. The default behavior of the ORB is to wait indefinitely.

You can set a couple of properties to avoid this condition:

- `com.ibm.CORBA.LocateRequestTimeout`
- `com.ibm.CORBA.RequestTimeout`

When the property `com.ibm.CORBA.enableLocateRequest` is set to true (the default is false), the ORB first sends a short message to the server to find the object that it needs to access. This first contact is the Locate Request. You must now set the `LocateRequestTimeout` to a value other than 0 (which is equivalent to infinity). A good value could be something around 5000 ms.

Also, set the `RequestTimeout` to a value other than 0. Because a reply to a request is often large, allow more time for the reply, such as 10,000 ms. These values are suggestions and might be too low for slow connections. When a request runs out of time, the client receives an explanatory CORBA exception.

When an application hangs, consider also another property that is called `com.ibm.CORBA.FragmentTimeout`. This property was introduced in IBM ORB 1.3.1, when the concept of fragmentation was implemented to increase performance. You can now split long messages into small chunks or fragments and send one after the other over the net. The ORB waits for 30 seconds (default value) for the next fragment before it throws an exception. If you set this property, you disable this timeout, and problems of waiting threads might occur.

If the problem seems to be a deadlock or hang, capture the Javacore 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 Javacore” on page 245.

In general, stop the application, enable the orb traces and restart the application. When the hang is reproduced, the partial traces that can be retrieved can be used by the IBM ORB service team to help understand where the problem is.

Running the client without the server running before the client is started

An example of the error messages that are generated from this process.

This operation outputs:

```
(org.omg.CORBA.COMM_FAILURE)
Hello Client exception:
org.omg.CORBA.COMM_FAILURE:minor code:1 completed:No
    at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:145)
    at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:77)
    at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
    at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:75)
    at com.ibm.rmi.corba.ClientDelegate.createRequest(ClientDelegate.java:440)
    at com.ibm.rmi.corba.ClientDelegate.is_a(ClientDelegate.java:571)
    at org.omg.CORBA.portable.ObjectImpl._is_a(ObjectImpl.java:74)
    at org.omg.CosNaming.NamingContextHelper._narrow(NamingContextHelper.java:58)
    at com.sun.jndi.cosnaming.CNCtx.callResolve(CNCtx.java:327)
```

Client and server are running, but not naming service

An example of the error messages that are generated from this process.

The output is:

```
Hello Client exception:Cannot connect to ORB
Javax.naming.CommunicationException:
    Cannot connect to ORB.Root exception is org.omg.CORBA.COMM_FAILURE minor code:1 completed:No
    at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:145)
    at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:77)
    at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
    at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:75)
    at com.ibm.rmi.corba.ClientDelegate.createRequest(ClientDelegate.java:440)
    at com.ibm.rmi.corba.InitialNamingClient.resolve(InitialNamingClient.java:197)
    at com.ibm.rmi.corba.InitialNamingClient.cachedInitialReferences(InitialNamingClient.j
    at com.ibm.rmi.corba.InitialNamingClient.resolve_initial_references(InitialNamingClie
    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.TRANIENT CONNECT_FAILURE)

Hello Client exception:Problem contacting address:corbaloc:iiop:machine2:2809/NameService
javax.naming.CommunicationException:Problem contacting address:corbaloc:iiop:machine2:2809/N
    is org.omg.CORBA.TRANIENT:CONNECT_FAILURE (1)minor code:4942F301 completed:No
    at com.ibm.CORBA.transport.TransportConnectionBase.connect(TransportConnectionBase.jav
    at com.ibm.rmi.transport.TCPTransport.getConnection(TCPTransport.java:178)
    at com.ibm.rmi.iiop.TransportManager.get(TransportManager.java:79)
    at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:131)
    at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
    at com.ibm.CORBA.iiop.ClientDelegate._createRequest(ClientDelegate.java:2096)
    at com.ibm.CORBA.iiop.ClientDelegate.createRequest(ClientDelegate.java:1264)
    at com.ibm.CORBA.iiop.ClientDelegate.createRequest(ClientDelegate.java:1177)
    at com.ibm.rmi.corba.InitialNamingClient.resolve(InitialNamingClient.java:252)
    at com.ibm.rmi.corba.InitialNamingClient.cachedInitialReferences(InitialNamingClient.j
    at com.ibm.rmi.corba.InitialNamingClient.resolve_initial_references(InitialNamingClie
    at com.ibm.rmi.corba.InitialReferenceClient.resolve_initial_references(InitialReferenc

```

```
at com.ibm.rmi.corba.ORB.resolve_initial_references(ORB.java:3211)
at com.ibm.rmi.iiop.ORB.resolve_initial_references(ORB.java:523)
at com.ibm.CORBA.iiop.ORB.resolve_initial_references(ORB.java:2898)
.....
```

IBM ORB service: collecting data

This section describes how to collect data about ORB problems.

If after all these verifications, the problem is still present, collect at all nodes of the problem the following:

- Operating system name and version.
- Output of `java -version`.
- Output of `java com.ibm.CORBA.iiop.Version`.
- Output of `rmic -iiop -version`, if `rmic` is involved.
- ASV build number (WebSphere Application Server only).
- If you think that the problem is a regression, include the version information for the most recent known working build and for the failing build.
- If this is a runtime problem, collect debug and communication traces of the failure from each node in the system (as explained earlier in this section).
- If the problem is in `rmic -iiop` or `rmic -idl`, set the options:
`-J-Djavac.dump.stack=1 -Xtrace`, and capture the output.
- Typically this step is not necessary. If it looks like the problem is in the buffer fragmentation code, IBM service will return the defect asking for an additional set of traces, which you can produce by executing with
`-Dcom.ibm.CORBA.FragmentSize=0`.

A testcase is not essential, initially. However, a working testcase that demonstrates the problem by using only the Java SDK classes will speed up the resolution time for the problem.

Preliminary tests

The ORB is affected by problems with the underlying network, hardware, and JVM.

When a problem occurs, the ORB can throw an `org.omg.CORBA.*` exception, some text that describes the reason, a minor code, and a completion status. Before you assume that the ORB is the cause of problem, ensure the following:

- The scenario can be reproduced in a similar configuration.
- The JIT is disabled (see Chapter 26, “JIT problem determination,” on page 317).

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.

Chapter 18. 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 208
- “Common NLS problem and possible causes” on page 209

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
```

On Windows platforms

Most text processing applications have a drop-down list of the available system fonts, or you can use the **Settings** → **Control Panel** → **Fonts** application.

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--*-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.

Font utilities on Windows systems

Windows has no built-in utilities similar to those offered by other platforms.

Common NLS problem and possible causes

A common NLS problem with potential solutions.

Why do I see a square box or ??? (question marks) in the SDK components?

This effect is caused mainly because Java is not able to find the correct font file to display the character. If a Korean character should be displayed, the system should be using the Korean locale, so that Java can take the correct font file. If you are seeing boxes or queries, check the following:

For AWT components:

1. Check your locale with `locale`.
2. To change the locale, export `LANG=zh_TW` (for example)
3. If this still does not work, try to log in with the required language.

For Swing components:

1. Check your locale with `locale`
2. To change the locale, export `LANG=zh_TW` (for example)
3. If you know which font you have used in your application, such as serif, try to get the corresponding physical font by looking in the fontpath. If the font file is missing, try adding it there.

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

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

Character not displayed in TextArea or TextField when using Motif

These components are Motif components (Linux and USS). Java gives a set of fonts to Motif to render the character. If the characters are not displayed properly, use the following Motif application to check whether the character is displayable by your Motif.

```
#include <stdio.h>
#include <locale.h>
#include <Xm/Xm.h>
#include <Xm/PushButton.h>
main(int argc, char **argv)
{
    XtAppContext    context;
    Widget toplevel, pushb;
    Arg    args[8];
    Cardinal    i, n;
    XmString    xmstr;
    char ptr[9];
    /* ptr contains the hex. Equivalent of unicode value */
    ptr[0] = 0xc4;    /*4E00*/
    ptr[1] = 0xa1;
    ptr[2] = 0xa4;    /*4E59*/
    ptr[3] = 0x41;
    ptr[4] = 0xa4;    /*4EBA*/
    ptr[5] = 0x48;
    ptr[6] = 0xa4;    /* 4E09 */
    ptr[7] = 0x54;
    ptr[8] = 0x00;
```

```

    setlocale(LC_ALL, "");
    toplevel = XtAppInitialize(&context, "", NULL, 0, &argc, argv,
                              NULL, NULL, 0);

    n=0;
    XtSetArg(args[n], XmNgeometry, "=225x225+50+50"); n++;
    XtSetArg(args[n], XmNallowShellResize, True); n++;
    XtSetValues(toplevel, args, n);
    xmstr =XmStringCreateLocalized(ptr);
    n=0;
    XtSetArg(args[n], XmNlabelString, xmstr); n++;
    pushb = XmCreatePushButton(toplevel, "PushB", args, n);
    XtManageChild(pushb);
    XtRealizeWidget(toplevel);
    XtAppMainLoop(context);
}
Compilation: cc -lXm -lXt -o motif motif.c

```

Chapter 19. Attach API problem determination

This section helps you solve problems involving the Attach API.

The IBM Java Attach API uses shared semaphores, sockets, and file system artifacts to implement the attach protocol. Problems with these artifacts might adversely affect the operation of applications when they use the attach API.

Note: Error messages from agents on the target VM go to stderr or stdout for the target VM. They are not reported in the messages output by the attaching VM.

Deleting files in /tmp

The attach API depends on the contents of a common directory. By default the common directory is /tmp/.com_ibm_tools_attach. Problems are caused if you modify the common directory in one of the following ways:

- Deleting the common directory.
- Deleting the contents of the common directory.
- Changing the permissions of the common directory or any of its content.

If you do modify the common directory, possible effects include:

- Semaphore “leaks” might occur, where excessive numbers of unused shared semaphores are opened. You can remove the semaphores using the command:
`ipcrm -s <semid>`

Use the command to delete semaphores that have keys starting with “0xa1”.

- The Java VMs might not be able to list existing target VMs.
- The Java VMs might not be able to attach to existing target VMs.
- The Java VM might not be able to enable its attach API.

If the common directory cannot be used, a Java VM attempts to recreate the common directory. However, the JVM cannot recreate the files related to currently executing VMs.

VirtualMachineDescriptor.displayName() returns the process ID, not the command line.

This result is a known limitation of the IBM Java 5 implementation. To specify the display name for a Java application, set the **com.ibm.tools.attach.displayName** property when you launch the application.

z/OS console messages reporting security violations in /tmp

The Attach API stores control files in the directory /tmp/.com_ibm_tools_attach. To prevent the display of security violation messages, use one of the following options:

- Add a security exception.
- Specify a different control directory, by setting the **com.ibm.tools.attach.directory** system property.

The VirtualMachine.attach(String id) method reports AttachNotSupportedException: No provider for virtual machine id

There are several possible reasons for this message:

- The target VM might be owned by another userid. The attach API can only connect a VM to a target VM with the same userid.
- The attach API for the target VM might not have launched yet. There is a short delay from when the Java VM launches to when the attach API is functional.
- The attach API for the target VM might have failed. Verify that the directory /tmp/.com_ibm_tools_attach/<id> exists, and that the directory is readable and writable by the userid.
- The target directory /tmp/.com_ibm_tools_attach/<id> might have been deleted.
- The attach API might not have been able to open the shared semaphore. To verify that there is at least one shared semaphore, use the command:
`ipcs -s`

If there is a shared semaphore, at least one key starting with "0xa1" appears in the output from the ipcs command.

Note: The number of available semaphores is limited on systems which use System V IPC, including Linux, z/OS, and AIX.

The VirtualMachine.attach() method reports AttachNotSupportedException

There are several possible reasons for this message:

- The target process is dead or suspended.
- The target process, or the hosting system is heavily loaded. The result is a delay in responding to the attach request.
- The network protocol has imposed a wait time on the port used to attach to the target. The wait time might occur after heavy use of the attach API, or other protocols which use sockets. To check if any ports are in the TIME_WAIT state, use the command:
`netstat -a`

The VirtualMachine.loadAgent(), VirtualMachine.loadAgentLibrary(), or VirtualMachine.loadAgentPath() methods report com.sun.tools.attach.AgentLoadException or com.sun.tools.attach.AgentInitializationException

There are several possible reasons for this message:

- The JVMTI agent or the agent JAR file might be corrupted. Try loading the agent at startup time using the -javaagent, -agentlib, or -agentpath option, depending on which method reported the problem.
- The agent might be attempting an operation which is not available after VM startup.

| **A process running as root can see a target using**
| **AttachProvider.listVirtualMachines(), but attempting to attach**
| **results in an AttachNotSupportedException**

| A process can attach only to processes owned by the same user. To attach to a
| non-root process from a root process, first use the su command to change the
| effective UID of the attaching process to the UID of the target UID, before
| attempting to attach.

Part 4. Using diagnostic tools

Diagnostics tools are available to help you solve your problems.

This section describes how to use the tools. The chapters are:

- Chapter 20, "Overview of the available diagnostics," on page 217
- Chapter 21, "Using dump agents," on page 223
- Chapter 22, "Using Javadump," on page 245
- Chapter 23, "Using Heapdump," on page 257
- Chapter 24, "Using system dumps and the dump viewer," on page 263
- Chapter 25, "Tracing Java applications and the JVM," on page 283
- Chapter 26, "JIT problem determination," on page 317
- Chapter 28, "Garbage Collector diagnostics," on page 329
- Chapter 29, "Class-loader diagnostics," on page 347
- Chapter 30, "Shared classes diagnostics," on page 351
- Chapter 31, "Using the Reliability, Availability, and Serviceability Interface," on page 371
- Chapter 32, "Using the HPROF Profiler," on page 385
- Chapter 33, "Using the JVMTI," on page 391
- Chapter 34, "Using the Diagnostic Tool Framework for Java," on page 393
- Chapter 35, "Using JConsole," on page 401

Note: JVMPI is now a deprecated interface, replaced by JVMTI.

Chapter 20. Overview of the available diagnostics

The diagnostics information that can be produced by the JVM is described in the following topics. A range of supplied tools can be used to post-process this information and help with problem determination.

Subsequent topics in this part of the *Information Center* give more details on the use of the information and tools in solving specific problem areas.

Some diagnostic information (such as that produced by Heapdump) is targeted towards specific areas of Java (classes and object instances in the case of Heapdumps), whereas other information (such as tracing) is targeted towards more general JVM problems.

Categorizing the problem

During problem determination, one of the first objectives is to identify the most probable area where the problem originates.

Many problems that seem to be a Java problem originate elsewhere. Areas where problems can arise include:

- The JVM itself
- Native code
- Java applications
- An operating system or system resource
- A subsystem (such as database code)
- Hardware

You might need different tools and different diagnostic information to solve problems in each area. The tools described here are (in the main) those built in to the JVM or supplied by IBM for use with the JVM. The majority of these tools are cross-platform tools, although there might be the occasional reference to other tools that apply only to a specific platform or varieties of that platform. Many other tools are supplied by hardware or system software vendors (such as system debuggers). Some of these tools are introduced in the platform-specific sections.

Summary of diagnostic information

A running IBM JVM includes mechanisms for producing different types of diagnostic data when different events occur.

In general, the production of this data happens under default conditions, but can be controlled by starting the JVM with specific options (such as **-Xdump**; see Chapter 21, “Using dump agents,” on page 223). Older versions of the IBM JVM controlled the production of diagnostic information through the use of environment variables. You can still use these environment variables, but they are not the preferred mechanism and are not discussed in detail here. Appendix B, “Environment variables,” on page 413 lists the supported environment variables).

The format of the various types of diagnostic information produced is specific to the IBM JVM and might change between releases of the JVM.

The types of diagnostic information that can be produced are:

Javadump

The Javadump is sometimes referred to as a Javacore or thread dump in some JVMs. This dump is in a human-readable format produced by default when the JVM terminates unexpectedly because of an operating system signal, an `OutOfMemoryError` exception, or when the user enters a reserved key combination (for example, **Ctrl-Break** on Windows). It can also be generated by calling `com.ibm.jvm.Dump.JavaDump()` from inside the application. A Javadump summarizes the state of the JVM at the instant the signal occurred. Much of the content of the Javadump is specific to the IBM JVM. See Chapter 22, “Using Javadump,” on page 245 for details.

Heapdump

The JVM can generate a Heapdump at the request of the user (for example by calling `com.ibm.jvm.Dump.HeapDump()` from inside the application) or (by default) when the JVM terminates because of an `OutOfMemoryError` exception. You can specify finer control of the timing of a Heapdump with the **-Xdump:heap** option. For example, you could request a Heapdump after a certain number of full garbage collections have occurred. The default Heapdump format (phd files) is not human-readable and you process it using available tools such as Heaproots. See Chapter 23, “Using Heapdump,” on page 257 for more details.

System dumps

System dumps (also known as core dumps on Linux platforms) are platform-specific files that contain information about the active processes, threads, and system memory. System dumps are usually large. By default, system dumps are produced by the JVM only when the JVM fails unexpectedly because of a GPF (general protection fault) or a major JVM or system error. You can also request a system dump by calling `com.ibm.jvm.Dump.SystemDump()` from your application. You can use the **-Xdump:system** option to produce system dumps when other events occur.

Garbage collection data

A JVM started with the **-verbose:gc** option produces output in XML format that can be used to analyze problems in the Garbage Collector itself or problems in the design of user applications. Numerous other options affect the nature and amount of Garbage Collector diagnostic information produced. See Chapter 28, “Garbage Collector diagnostics,” on page 329 for more information.

Trace data

The IBM JVM tracing allows execution points in the Java code and the internal JVM code to be logged. The **-Xtrace** option allows the number and areas of trace points to be controlled, as well as the size and nature of the trace buffers maintained. The internal trace buffers at a time of failure are also available in a system dump and tools are available to extract them from a system dump. Generally, trace data is written to a file in an encoded format and then a trace formatter converts the data into a readable format. However, if small amounts of trace are to be produced and performance is not an issue, trace can be routed to `STDERR` and will be pre-formatted. For more information, see Chapter 25, “Tracing Java applications and the JVM,” on page 283.

Other data

Special options are available for producing diagnostic information relating to

- The JIT (see Chapter 26, “JIT problem determination,” on page 317)
- Class loading (see Chapter 29, “Class-loader diagnostics,” on page 347)
- Shared classes (see Chapter 30, “Shared classes diagnostics,” on page 351)

The SDK includes a JVMTI based profiling tool called HPROF, which produces information that can help you to determine the parts of an application that might be using system resources; see Chapter 32, “Using the HPROF Profiler,” on page 385 for more details.

The SDK also includes an unsupported, experimental tool called JConsole. This graphical monitoring tool is based on the `java.lang.management` API, which you can use to observe and, for some properties, control various aspects of the JVM's behavior.

Summary of cross-platform tooling

IBM has several cross-platform diagnostic tools. The following sections provide brief descriptions of the tools and indicate the different areas of problem determination to which they are suited.

Heapdump analysis tooling

A number of tools are available for working with Heapdumps.

See Chapter 23, “Using Heapdump,” on page 257 for more information.

Cross-platform dump viewer

The cross-system dump viewer uses the dump files that the operating system generates to resolve data relevant to the JVM.

This tool is provided in two parts:

1. `jextract` - platform-specific utility to extract and package (compress) data from the dump generated by the native operating system
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. It is present only in the IBM SDK for Java. It is cross-platform and allows you to perform useful dump analysis without the need for a workstation or operating system of the type on which the problem was produced or knowledge of the system debugger on the relevant platform.

For more information, see Chapter 24, “Using system dumps and the dump viewer,” on page 263.

JVMTI tools

The JVMTI (JVM Tool Interface) is a programming interface for use by tools. It replaces the Java Virtual Machine Profiler Interface (JVMPPI) and the Java Virtual Machine Debug Interface (JVMDI).

For information on the JVMTI, see Chapter 33, “Using the JVMTI,” on page 391. The HPROF tool provided with the SDK has been updated to use the JVMTI; see Chapter 32, “Using the HPROF Profiler,” on page 385.

JVMPI tools

The JVMPI is now a deprecated interface. Use JVMTI instead.

JVMPI was officially described by Sun as “an experimental interface for profiling”. Now that it is a deprecated interface, you are advised to upgrade existing tools to use the JVMTI (Java Virtual Machine Tool Interface), described in Chapter 33, “Using the JVMTI,” on page 391. An article to help you with the upgrade is at:

<http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition/>

The IBM JVM still supports the deprecated JVMPI specification. Existing JVMPI-based tools (such as the vendor tools JProbe, OptimizeIt, TrueTime, and Quantify®) that use the JVMPI should continue to work. The IBM SDK provided tool HPROF has been updated to use the JVMTI; see Chapter 32, “Using the HPROF Profiler,” on page 385.

JPDA tools

The Java Platform Debugging Architecture (JPDA) is a common standard for debugging JVMs. The IBM Virtual Machine for Java is fully JPDA compatible.

Any JPDA debugger can be attached to the IBM Virtual Machine for Java. Because they are debuggers, JPDA tools are best suited to tracing application problems that have repeatable conditions, such as:

- Memory leaks in applications.
- Unexpected termination or “hanging”.

An example of a JPDA tool is the debugger that is bundled with Eclipse for Java.

DTFJ

The Diagnostic Tool Framework for Java (DTFJ) is a Java application programming interface (API) from IBM used to support the building of Java diagnostics tools.

You process the dumps passed to DTFJ with the jextract tool; see “Using the dump extractor, jextract” on page 264. The jextract tool produces metadata from the dump, which allows the internal structure of the JVM to be analyzed. jextract must be run on the system that produced the dump.

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 Chapter 34, “Using the Diagnostic Tool Framework for Java,” on page 393.

Trace formatting

JVM trace is a key diagnostic tool for the JVM. The IBM JVM incorporates a large degree of flexibility in determining what is traced and when it is traced. This flexibility enables you to tailor trace so that it has a relatively small effect on performance.

The IBM Virtual Machine for Java contains many embedded trace points. **In this release, maximal tracing is switched on by default for a few level 1 tracepoints**

and exception trace points. Command-line options allow you to set exactly what is to be traced, and specify where the trace output is to go. Trace output is generally in an encoded format and requires a trace formatter to be viewed successfully.

In addition to the embedded trace points provided in the JVM code, you can place your own application trace points in your Java code. You can activate tracing for entry and exit against all methods in all classes. Alternatively, you can activate tracing for a selection of methods in a selection of classes. Application and method traces are interleaved in the trace buffers with the JVM embedded trace points. The tracing allows detailed analysis of the routes taken through the code.

Tracing is used mainly for performance and leak problem determination. Trace data might also provide clues to the state of a JVM before an unexpected termination or “hang”.

Trace and trace formatting are IBM-specific; that is, they are present only in the IBM Virtual Machine for Java. See “Using method trace” on page 309 and Chapter 25, “Tracing Java applications and the JVM,” on page 283 for more details. Although trace is not easy to understand, it is an effective tool.

JVMRI

The JVMRI interface will be deprecated in the near future and replaced by JVMTI extensions.

The JVMRI (JVM RAS Interface, where RAS stands for Reliability, Availability, Serviceability) allows you to control several JVM operations programmatically.

For example, the IBM Virtual Machine for Java contains a large number of embedded trace points. Most of these trace points are switched off by default. A JVMRI agent can act as a Plug-in to allow real-time control of trace information. You use the **-Xrun** command-line option so that the JVM itself loads the agent at startup. When loaded, a JVMRI agent can dynamically switch individual JVM trace points on and off, control the trace level, and capture the trace output.

The JVMRI is particularly useful when applied to performance and leak problem determination, although the trace file might provide clues to the state of a JVM before an unexpected termination or hang.

The RAS Plug-in interface is an IBM-specific interface; that is, it is present only in the IBM Virtual Machine for Java. See Chapter 31, “Using the Reliability, Availability, and Serviceability Interface,” on page 371 for details. You need some programming skills and tools to be able to use this interface.

Chapter 21. Using dump agents

Dump agents are set up during JVM initialization. They enable you to use events occurring in the JVM, such as Garbage Collection, thread start, or JVM termination, to initiate dumps or to start an external tool.

The default dump agents are sufficient for most cases. Use the **-Xdump** option to add and remove dump agents for various JVM events, update default dump settings (such as the dump name), and limit the number of dumps that are produced.

This section describes:

- “Using the -Xdump option”
- “Dump agents” on page 227
- “Dump events” on page 232
- “Advanced control of dump agents” on page 233
- “Dump agent tokens” on page 238
- “Default dump agents” on page 238
- “Removing dump agents” on page 239
- “Dump agent environment variables” on page 239
- “Signal mappings” on page 241
- “Windows, Linux, AIX, and i5/OS specifics” on page 241
- “z/OS specifics” on page 242

Using the -Xdump option

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

The **-Xdump** option allows you to:

- Add and remove dump agents for various JVM events.
- Update default dump agent settings.
- Limit the number of dumps produced.
- Show dump agent help.

You can have multiple **-Xdump** options on the command line and also multiple dump types triggered by multiple events. For example:

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

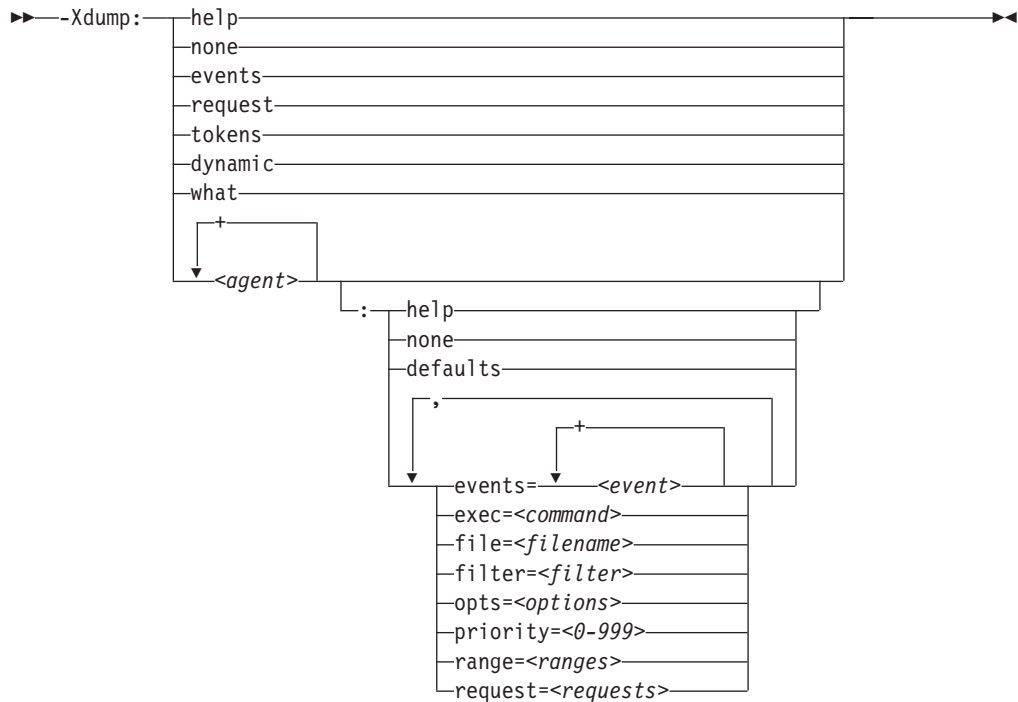
turns off all Heapdumps and create a dump agent that produces a Heapdump and a Javadump when either a vmstart or vmstop event occurs.

You can use the **-Xdump:what** option to list the registered dump agents. The registered dump agents listed might be different to those specified because the JVM ensures that multiple **-Xdump** options are merged into a minimum set of dump agents.

The events keyword is used as the prime trigger mechanism. However, you can use additional keywords to further control the dump produced.

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

-Xdump command-line option syntax



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

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

For more information, see the manual for your shell.

Help options

These options display usage and configuration information for dumps, as shown in the following table:

Command	Result
-Xdump:help	Display general dump help
-Xdump:events	List available trigger events
-Xdump:request	List additional VM requests
-Xdump:tokens	List recognized label tokens
-Xdump:what	Show registered agents on startup
-Xdump:<agent>:help	Display detailed dump agent help
-Xdump:<agent>:defaults	Display default settings for this agent

Merging -Xdump agents

-Xdump agents are always merged internally by the JVM, as long as none of the agent settings conflict with each other.

If you configure more than one dump agent, each responds to events according to its configuration. However, the internal structures representing the dump agent configuration might not match the command line, because dump agents are merged for efficiency. Two sets of options can be merged as long as none of the agent settings conflict. This means that the list of installed dump agents and their parameters produced by **-Xdump:what** might not be grouped in the same way as the original **-Xdump** options that configured them.

For example, you can use the following command to specify that a dump agent collects a javadump on class unload:

```
java -Xdump:java:events=unload -Xdump:what
```

This command does not create a new agent, as can be seen in the results from the **-Xdump:what** option.

Windows:

```
...
-----
-Xdump:java:
  events=gpf+user+abort+unload,
  label=C:\javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
  range=1..0,
  priority=10,
  request=exclusive
-----
```

Other platforms:

```
...
-----
-Xdump:java:
  events=gpf+user+abort+unload,
  label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
  range=1..0,
  priority=10,
  request=exclusive
-----
```

The configuration is merged with the existing javadump agent for events **gpf**, **user**, and **abort**, because none of the specified options for the new unload agent conflict with those for the existing agent.

In the above example, if one of the parameters for the unload agent is changed so that it conflicts with the existing agent, then it cannot be merged. For example, the following command specifies a different priority, forcing a separate agent to be created:

```
java -Xdump:java:events=unload,priority=100 -Xdump:what
```

The results of the **-Xdump:what** option in the command are as follows.

Windows:

```
...
-----
-Xdump:java:
  events=unload,
```

```

label=C:\javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
range=1..0,
priority=100,
request=exclusive
-----
-Xdump:java:
  events=gpf+user+abort,
  label=C:\javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
  range=1..0,
  priority=10,
  request=exclusive
-----

```

Other platforms:

```

...
-----
-Xdump:java:
  events=unload,
  label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
  range=1..0,
  priority=100,
  request=exclusive
-----
-Xdump:java:
  events=gpf+user+abort,
  label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
  range=1..0,
  priority=10,
  request=exclusive
-----

```

To merge dump agents, the **request**, **filter**, **opts**, **label**, and **range** parameters must match exactly. If you specify multiple agents that filter on the same string, but keep all other parameters the same, the agents are merged. For example:

```

java -Xdump:none -Xdump:java:events=uncaught,filter=java/lang/NullPointerException \
-Xdump:java:events=unload,filter=java/lang/NullPointerException -Xdump:what

```

The results of this command are as follows.

Windows:

Registered dump agents

```

-----
-Xdump:java:
  events=unload+uncaught,
  filter=java/lang/NullPointerException,
  label=C:\javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
  range=1..0,
  priority=10,
  request=exclusive
-----

```

Other platforms:

Registered dump agents

```

-----
-Xdump:java:
  events=unload+uncaught,
  filter=java/lang/NullPointerException,
  label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt,
  range=1..0,
  priority=10,
  request=exclusive
-----

```

Before Service Refresh 7, it was possible to merge filtered and non-filtered agents. For example, to configure a filter agent to trigger a javadump when a `NullPointerException` is not caught, and to merge the agent with an existing **gpf**, **abort**, and **user** javadump agent, run the following command:

```
java -Xdump:none -Xdump:java:events=gpf+abort+user \\  
-Xdump:java:events=uncaught,filter=java/lang/NullPointerException -Xdump:what
```

The output from this command will be similar to the following:

```
Registered dump agents  
-----  
dumpFn=doJavaDump  
events=gpf+user+abort+uncaught  
filter=java/lang/NullPointerException  
label=javacore.%Y%m%d.%H%M%S.%pid.txt  
range=1..0  
priority=10  
request=exclusive  
opts=  
-----
```

The **gpf**, **user**, and **abort** events do not support filtering. This means that before Service Refresh 7, the filter is ignored for these events.

From Service Refresh 7, this kind of merging is not performed. Using Service Refresh 7 or later, if you run the following command, a separate agent will be created for the uncaught event:

```
java -Xdump:none -Xdump:java:events=gpf+abort+user \\  
-Xdump:java:events=uncaught,filter=java/lang/NullPointerException -Xdump:what
```

The output from this command will be similar to the following:

```
Registered dump agents  
-----  
dumpFn=doJavaDump  
events=gpf+user+abort  
filter=  
label=C:\javacore.%Y%m%d.%H%M%S.%pid.%seq.txt  
range=1..0  
priority=10  
request=exclusive  
opts=  
-----  
dumpFn=doJavaDump  
events=uncaught  
filter=java/lang/NullPointerException  
label=C:\javacore.%Y%m%d.%H%M%S.%pid.%seq.txt  
range=1..0  
priority=10  
request=exclusive  
opts=  
-----
```

Dump agents

A dump agent performs diagnostic tasks when triggered. Most dump agents save information on the state of the JVM for later analysis. The “tool” agent can be used to trigger interactive diagnostics.

The following table shows the dump agents:

Dump agent	Description
------------	-------------

stack	Stack dumps are very basic dumps in which the status and Java stack of the thread is written to stderr. This agent is available from Java 5 SR10 onwards. See "Stack dumps" on page 229.
console	Basic thread dump to stderr.
system	Capture raw process image. See Chapter 24, "Using system dumps and the dump viewer," on page 263.
tool	Run command-line program.
java	Write application summary. See Chapter 22, "Using Javacore," on page 245.
heap	Capture heap graph. See Chapter 23, "Using Heapdump," on page 257.
snap	Take a snap of the trace buffers.

Console dumps

Console dumps are very basic dumps, in which the status of every Java thread is written to stderr.

In this example, the **range=1..1** suboption is used to control the amount of output to just one thread start (in this case, the start of the Signal Dispatcher thread).

```
java -Xdump:console:events=thrstart+thrstop,range=1..1
```

```
JVMDUMP006I Processing Dump Event "thrstart", detail "" - Please Wait.
----- Console dump -----
```

Stack Traces of Threads:

```
ThreadName=Signal Dispatcher(0805BFFC)
Status=Running
```

```
ThreadName=main(0805B5FC)
Status=Waiting
Monitor=0805ADE0 (Thread public flags mutex)
Count=0
Owner=(00000000)
  In com/ibm/oti/vm/BootstrapClassLoader.loadClass(Ljava/lang/String;)Ljava/lang/Class;
  In com/ibm/misc/SystemInitialization.lastChanceHook()V
  In java/lang/System.completeInitialization()V
  In java/lang/Thread.<init>(Ljava/lang/String;Ljava/lang/Object;IZ)V
```

```
~~~~~ Console dump ~~~~~
JVMDUMP013I Processed Dump Event "thrstart", detail "".
```

Two threads are displayed in the dump because the main thread does not generate a thrstart event.

System dumps

System dumps involve dumping the address space and as such are generally very large.

The bigger the footprint of an application the bigger its dump. A dump of a major server-based application might take up many gigabytes of file space and take several minutes to complete. In this example, the file name is overridden from the default.

Windows:

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

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

```
JVMDUMP006I Processing Dump Event "vmstop", detail "#00000000" - Please Wait.  
JVMDUMP007I JVM Requesting System Dump using 'C:\sdk\sdk\jre\bin\my.dmp'  
JVMDUMP010I System Dump written to C:\sdk\sdk\jre\bin\my.dmp  
JVMDUMP013I Processed Dump Event "vmstop", detail "#00000000".
```

Other platforms:

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

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

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

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

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

See Chapter 24, “Using system dumps and the dump viewer,” on page 263 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. Stack dumps are available from Java 5 SR 10 onwards.

In the following example, the main thread has allocated a byte array of size 1549128 bytes:

```
JVMDUMP006I Processing dump event "allocation", detail "1549128 bytes, type byte[]" - please wait.  
Thread=main (0188701C) Status=Running  
    at sun/misc/Resource.getBytes() [B (Resource.java:109)  
    at java/net/URLClassLoader.defineClass(Ljava/lang/String;Lsun/misc/Resource;)Ljava/lang/Class;  
    at java/net/URLClassLoader.access$300(Ljava/net/URLClassLoader;Ljava/lang/String;Lsun/misc/Resource;)Ljava/lang/Class;  
    at java/net/URLClassLoader$ClassFinder.run()Ljava/lang/Object; (URLClassLoader.java:901)  
    at java/security/AccessController.doPrivileged(Ljava/security/PrivilegedExceptionAction;Ljava/lang/Runnable;)Ljava/lang/Object;  
    at java/net/URLClassLoader.findClass(Ljava/lang/String;)Ljava/lang/Class; (URLClassLoader.java:1000)  
    at java/lang/ClassLoader.loadClass(Ljava/lang/String;)Ljava/lang/Class; (ClassLoader.java:611)  
    at sun/misc/Launcher$AppClassLoader.loadClass(Ljava/lang/String;)Ljava/lang/Class; (Launcher.java:104)  
    at java/lang/ClassLoader.loadClass(Ljava/lang/String;)Ljava/lang/Class; (ClassLoader.java:611)  
    at TestLargeAllocations.main([Ljava/lang/String;)V (TestLargeAllocations.java:49)
```

LE CEEDUMPs

LE CEEDUMPs are a z/OS only formatted summary system dump that show stack traces for each thread that is in the JVM process, together with register information and a short dump of storage for each register.

This example of a traceback is taken from a CEEDUMP produced by a crash. The traceback shows that the crash occurred in the `rasTriggerMethod` method:

```
CEE3DMP V1 R8.0: CHAMBER.JVM.TDUMP.CHAMBER.D080910.T171047
```

09/10/08 5:10:5

Traceback:

DSA	Addr	Program	Unit	PU	Addr	PU	Offset	Entry	E	Addr	E	Offset	Statement	LOC
-----	------	---------	------	----	------	----	--------	-------	---	------	---	--------	-----------	-----

.....	124222A8	CEEHDSP	07310AF0	+00000CEC	CEEHDSP	07310AF0	+00000CEC	CEEP
	12421728	CEEHRNUH	0731F728	+00000092	CEEHRNUH	0731F728	+00000092	CEEP
	128461E0		12AB6A00	+0000024C	rasTriggerMethod			
					12AB6A00	+0000024C		2120 *PAT
	12846280		12AACBE8	+00000208	hookMethodEnter			
					12AACBE8	+00000208		1473 *PAT
	12846300		12A547C0	+000000B8	J9HookDispatch			
					12A547C0	+000000B8		157 *PAT
	12846380		12943840	+00000038	triggerMethodEnterEvent			
					12943840	+00000038		110 *PAT
.....								

When a CEEDUMP is produced by the JVM, the following message is issued:

```
JVMDUMP010I CEE dump written to /u/test/CEEDUMP.20090622.133914.65649
```

On 32-bit z/OS, if more than one CEEDUMP is produced during the lifetime of a JVM instance, the second and subsequent CEEDUMPs will be appended to the same file. The JVMDUMP010I messages will identify the same file each time.

On 64-bit z/OS, if more than one CEEDUMP is produced a separate CEEDUMP file is written each time, and the JVMDUMP010I messages will identify the separate files.

See Understanding the Language Environment dump in the *z/OS: Language Environment Debugging Guide* for more information.

Tool option

The **tool** option allows external processes to be started when an event occurs.

The following example displays a simple message when the JVM stops. The %pid token is used to pass the pid of the process to the command. The list of available tokens can be printed with **-Xdump:tokens**, or found in “Dump agent tokens” on page 238. If you do not specify a tool to use, a platform specific debugger is started.

Windows:

```
java -Xdump:tool:events=vmstop,exec="cmd /c echo %pid has finished"
-Xdump:tool:events=vmstart,exec="cmd /c echo %pid has started"
```

```
JVMDUMP006I Processing Dump Event "vmstart", detail "" - Please Wait.
JVMDUMP007I JVM Requesting Tool Dump using 'cmd /c echo 2184 has started'
JVMDUMP011I Tool Dump spawned process 2160
2184 has started
JVMDUMP013I Processed Dump Event "vmstart", detail "".
```

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

```
JVMDUMP006I Processing Dump Event "vmstop", detail "#00000000" - Please Wait.
JVMDUMP007I JVM Requesting Tool Dump using 'cmd /c echo 2184 has finished'
JVMDUMP011I Tool Dump spawned process 2204
2184 has finished
JVMDUMP013I Processed Dump Event "vmstop", detail "#00000000".
```

Other platforms:

```
java -Xdump:tool:events=vmstop,exec="echo process %pid has finished" -version
```

```
VMDUMP006I Processing dump event "vmstop", detail "#00000000" - please wait.
JVMDUMP007I JVM Requesting Tool dump using 'echo process 254050 has finished'
```

```
JVMDUMP011I Tool dump spawned process 344292
process 254050 has finished
JVMDUMP013I Processed dump event "vmstop", detail "#00000000".
```

By default, the **range** option is set to 1..1. If you do not specify a range option for the dump agent the tool will be started once only. To start the tool every time the event occurs, set the **range** option to 1..0. See “range option” on page 236 for more information.

Javadumps

Javadumps are an internally generated and formatted analysis of the JVM, giving information that includes the Java threads present, the classes loaded, and heap statistics.

An example of producing a Jav_dump when a class is loaded is shown below.

Windows:

```
java -Xdump:java:events=load,filter=*String
```

```
JVMDUMP006I Processing Dump Event "load", detail "java/lang/String" - Please Wait.
JVMDUMP007I JVM Requesting Java Dump using
C:\sdk\jre\bin\javacore.20051012.162700.2836.txt'
JVMDUMP010I Java Dump written to
C:\sdk\jre\bin\javacore.20051012.162700.2836.txt
JVMDUMP013I Processed Dump Event "load", detail "java/lang/String".
```

Other platforms:

```
java -Xdump:java:events=load,filter=java/lang/String -version
```

```
JVMDUMP006I Processing dump event "load", detail "java/lang/String" - please wait.
JVMDUMP007I JVM Requesting Java dump using '/home/user/javacore.20090602.094449.274632.0001.txt'
JVMDUMP010I Java dump written to /home/user/javacore.20090602.094449.274632.0001.txt
JVMDUMP013I Processed dump event "load", detail "java/lang/String".
```

See Chapter 22, “Using Jav_dump,” on page 245 for more information about analyzing a Jav_dump.

Heapdumps

Heapdumps produce phd format files by default.

Chapter 23, “Using Heapdump,” on page 257 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.

Windows:

```
java -Xdump:none -Xdump:heap:events=vmstop,opts=PHD+CLASSIC
```

```
JVMDUMP006I Processing Dump Event "vmstop", detail "#00000000" - Please Wait.
JVMDUMP007I JVM Requesting Heap Dump using
'C:\sdk\jre\bin\heapdump.20050323.142011.3272.phd'
JVMDUMP010I Heap Dump written to
C:\sdk\jre\bin\heapdump.20050323.142011.3272.phd
JVMDUMP007I JVM Requesting Heap Dump using
'C:\sdk\jre\bin\heapdump.20050323.142011.3272.txt'
JVMDUMP010I Heap Dump written to
C:\sdk\jre\bin\heapdump.20050323.142011.3272.txt
JVMDUMP013I Processed Dump Event "vmstop", detail "#00000000".
```

Other platforms:

java -Xdump:heap:events=vmstop,opts=PHD+CLASSIC -version

```
JVMDUMP006I Processing dump event "vmstop", detail "#00000000" - please wait.
JVMDUMP007I JVM Requesting Heap dump using '/home/user/heapdump.20090602.095239.164050.0001.phd'
JVMDUMP010I Heap dump written to /home/user/heapdump.20090602.095239.164050.0001.phd
JVMDUMP007I JVM Requesting Heap dump using '/home/user/heapdump.20090602.095239.164050.0001.txt'
JVMDUMP010I Heap dump written to /home/user/heapdump.20090602.095239.164050.0001.txt
JVMDUMP013I Processed dump event "vmstop", detail "#00000000".
```

See Chapter 23, “Using Heapdump,” on page 257 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 example below shows the production of a snap trace.

Windows:

java -Xdump:none -Xdump:snap:events=vmstop+vmstart

```
JVMDUMP006I Processing Dump Event "vmstart", detail "" - Please Wait.
JVMDUMP007I JVM Requesting Snap Dump using
'C:\sdk\jre\bin\Snap0001.20051012.161706.2804.trc'
JVMDUMP010I Snap Dump written to
C:\sdk\jre\bin\Snap0001.20051012.161706.2804.trc
JVMDUMP013I Processed Dump Event "vmstart", detail "".
```

```
Usage: java [-options] class [args...]
        (to execute a class)
====  extraneous lines removed for terseness  ====
-assert  print help on assert options
```

```
JVMDUMP006I Processing Dump Event "vmstop", detail "#00000000" - Please Wait.
JVMDUMP007I JVM Requesting Snap Dump using
'C:\sdk\jre\bin\Snap0002.20051012.161706.2804.trc'
JVMDUMP010I Snap Dump written to
C:\sdk\jre\bin\Snap0002.20051012.161706.2804.trc
JVMDUMP013I Processed Dump Event "vmstop", detail "#00000000".
```

Other platforms:

java -Xdump:none -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".
```

By default snap traces are given sequential numbers (Snap0001 then Snap0002). Snap traces require the use of the trace formatter for further analysis.

See “Using the trace formatter” on page 304 for more information about analyzing a snap trace.

Dump events

Dump agents are triggered by events occurring during JVM operation.

Some events can be filtered to improve the relevance of the output. See “filter option” on page 234 for more information.

The table below shows events available as dump agent triggers:

Event	Triggered when...	Filter operation
gpf	A General Protection Fault (GPF) occurs.	
user	The JVM receives the SIGQUIT (Linux, AIX, z/OS, and i5/OS) or SIGBREAK (Windows) signal from the operating system.	
abort	The JVM receives the SIGABRT signal from the operating system.	
vmstart	The virtual machine is started.	
vmstop	The virtual machine stops.	Filters on exit code; for example, filter=#129..#192#-42#255
load	A class is loaded.	Filters on class name; for example, filter=java/lang/String
unload	A class is unloaded.	
throw	An exception is thrown.	Filters on exception class name; for example, filter=java/lang/OutOfMem*
catch	An exception is caught.	Filters on exception class name; for example, filter=*Memory*
uncaught	A Java exception is not caught by the application.	Filters on exception class name; for example, filter=*MemoryError
systhrow	A Java exception is about to be thrown by the JVM. This is different from the 'throw' event because it is only triggered for error conditions detected internally in the JVM.	Filters on exception class name; for example, filter=java/lang/OutOfMem*
thrstart	A new thread is started.	
blocked	A thread becomes blocked.	
thrstop	A thread stops.	
fullgc	A garbage collection cycle is started.	
slow	A thread takes longer than 50ms to respond to an internal JVM request.	
allocation	A Java object is allocated with a size matching the given filter specification	Filters on object size; a filter must be supplied. For example, filter=#5m will trigger on objects larger than 5 Mb. Ranges are also supported; for example, filter=#256k..512k will trigger on objects between 256 Kb and 512 Kb in size. This dump event is available from Java 5 SR 10 onwards.

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 230 for an example and usage information.

file option

The file option is used by dump agents that write to a file.

It specifies where the diagnostics information should be written. For example:

```
java -Xdump:heap:events=vmstop,file=my.dmp
```

When producing system dumps on z/OS platforms, use the dsn option instead of the file option. For example:

```
java -Xdump:system:events=vmstop,dsn=%uid.MYDUMP
```

You can use tokens to add context to dump file names. See “Dump agent tokens” on page 238 for more information.

The location for the dump is selected from these options, in this order:

1. The location specified on the command line.
2. The location specified by the relevant environment variable.
 - **IBM_JAVACOREDIR** for Javadump. **_CEE_DMPTARG** on z/OS.
 - **IBM_HEAPDUMPDIR** for Heapdump. **_CEE_DMPTARG** on z/OS.
 - **IBM_COREDIR** for system dump, **JAVA_DUMP_TDUMP_PATTERN** on z/OS.
 - **IBM_COREDIR** for snap traces, **_CEE_DMPTARG** on z/OS.
3. The current working directory of the JVM process.

If the directory does not exist, it will be created.

If the dump cannot be written to the selected location, the JVM will fall-back to the following locations, in this order:

1. On Windows platforms only, the system default location is C:\WINDOWS.
2. The location specified by the **TMPDIR** environment variable.
3. The /tmp directory.
4. C:\Temp on Windows platforms.

filter option

Some JVM events occur thousands of times during the lifetime of an application. Dump agents can use filters and ranges to avoid excessive dumps being produced.

Wildcards

You can use a wildcard in your exception event filter by placing an asterisk only at the beginning or end of the filter. The following command does not work because the second asterisk is not at the end:

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#.myVirtualMethod
```

In order to make this filter work, it must be changed to:

```
-Xdump:java:events=vmstop,filter=*InvalidArgumentException#MyApplication.*
```

Class loading and exception events

You can filter class loading (load) and exception (throw, catch, uncaught, systhrow) events by Java class name:

```
-Xdump:java:events=throw,filter=java/lang/OutOfMem*
-Xdump:java:events=throw,filter=*MemoryError
-Xdump:java:events=throw,filter=*Memory*
```

From Java 5 SR 9, you can filter throw, uncaught, and systhrow exception events by Java method name:

```
-Xdump:java:events=throw,filter=ExceptionClassName[#ThrowingClassName.throwingMethodName[#stackFrameIndex]]
```

Optional portions are shown in square brackets.

From Java 5 SR 9, you can filter the catch exception events by Java method name:

```
-Xdump:java:events=catch,filter=ExceptionClassName[#CatchingClassName.catchingMethodName]
```

Optional portions are shown in square brackets.

vmstop event

You can filter the JVM shut down event by using one or more exit codes:

```
-Xdump:java:events=vmstop,filter=#129..192#-42#255
```

slow event

From Java 5 SR 6, you can filter the slow event to change the time threshold from the default of 50 ms:

```
-Xdump:java:events=slow,filter=#300ms
```

You cannot set the filter to a time lower than the default time.

allocation event

You must filter the allocation event to specify the size of objects that cause a trigger. You can set the filter size from zero up to the maximum value of a 32 bit pointer on 32 bit platforms, or the maximum value of a 64 bit pointer on 64 bit platforms. Setting the lower filter value to zero triggers a dump on all allocations.

For example, to trigger dumps on allocations greater than 5 Mb in size, use:

```
-Xdump:stack:events=allocation,filter=#5m
```

To trigger dumps on allocations between 256Kb and 512Kb in size, use:

```
-Xdump:stack:events=allocation,filter=#256k..512k
```

The allocation event is available from Java 5 SR 10 onwards.

Other events

If you apply a filter to an event that does not support filtering, the filter is ignored.

opts option

The Heapdump agent uses this option to specify the type of file to produce. On z/OS, the system dump agent uses this option to specify the type of dump to produce.

Heapdumps and the opts option

You can specify a PHD Heapdump, a classic text Heapdump, or both. For example:

-Xdump:heap:opts=PHD (default)
-Xdump:heap:opts=CLASSIC
-Xdump:heap:opts=PHD+CLASSIC

See “Enabling text formatted (“classic”) Heapdumps” on page 258 for more information.

z/OS System dumps and the opts option

You can specify a system transaction dump (IEATDUMP), an LE dump (CEEDUMP), or both. For example:

-Xdump:system:opts=IEATDUMP (default)
-Xdump:system:opts=CEEDUMP
-Xdump:system:opts=IEATDUMP+CEEDUMP

The ceedump agent is the preferred way to specify LE dumps, for example:

-Xdump:ceedump:events=gpf

Priority option

One event can generate multiple dumps. The agents that produce each dump run sequentially and their order is determined by the priority keyword set for each agent.

Examination of the output from **-Xdump:what** shows that a gpf event produces a snap trace, a Javadump, and a system dump. In this example, the system dump will run first (priority 999), the snap dump second (priority 500), and the Javadump last (priority 10):

-Xdump:heap:events=vmstop,priority=123

The maximum value allowed for priority is 999. Higher priority dump agents will be started first.

If you do not specifically set a priority, default values are taken based on the dump type. The default priority and the other default values for a particular type of dump, can be displayed by using **-Xdump:<type>:defaults**. For example:

java -Xdump:heap:defaults -version

Default -Xdump:heap settings:

```
events=gpf+user
filter=
file=/home/user/heapdump.%Y%m%d.%H%M%S.%pid.phd
range=1..0
priority=40
request=exclusive+prewalk
opts=PHD
```

range option

You can start and stop dump agents on a particular occurrence of a JVM event by using the range suboption.

For example:

-Xdump:java:events=fullgc,range=100..200

Note: **range=1..0** against an event means "on every occurrence".

The JVM default dump agents have the **range** option set to 1..0 for all events except `systraw`. All `systraw` 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 238

If you add a new dump agent and do not specify the range, a default of 1..0 is used.

request option

Use the request option to ask the JVM to prepare the state before starting the dump agent.

The available options are listed in the following table:

Option value	Description
exclusive	Request exclusive access to the JVM.
compact	Run garbage collection. This option removes all unreachable objects from the heap before the dump is generated.
prewalk	Prepare the heap for walking.
serial	Suspend other dumps until this one has finished.

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 runtime. For example, you can direct Java dump files into a separate directory for each process, and guarantee unique files by adding a sequence number to the file name using:

-Xdump:java:defaults:file=dumps/%pid/javacore-%seq.txt

Or, for example, on z/OS, you can add the jobname to the Java dump file name using:

-Xdump:java:defaults:file=javacore.%job.%H%M%S.txt

This option does not add a Javacore agent; it updates the default settings for Javacore agents. Further Javacore 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 Javacore, that will change the file name used by the default Javacore agents. However, changing the default **range** option will not change the range used by the default Javacore agents, because those agents override the **range** option with specific values.

Dump agent tokens

Use tokens to add context to dump file names and to pass command-line arguments to the tool agent.

The tokens available are listed in the following table:

Token	Description
%Y	Year (4 digits)
%y	Year (2 digits)
%m	Month (2 digits)
%d	Day of the month (2 digits)
%H	Hour (2 digits)
%M	Minute (2 digits)
%S	Second (2 digits)
%pid	Process id
%uid	User name
%seq	Dump counter
%tick	msec counter
%home	Java home directory
%last	Last dump
%job	Job name (z/OS only)

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 239. for more information.

Use the **-Xdump:what** option on the command line to show the registered dump agents. The sample output shows the default dump agents that are in place:

```
java -Xdump:what
```

```
Registered dump agents
```

```
-----
```

```
dumpFn=doSystemDump
events=gpf+abort
filter=
label=/home/user/core.%Y%m%d.%H%M%S.%pid.%seq.dmp
range=1..0
priority=999
request=serial
opts=
```

```
-----
```

```
dumpFn=doSnapDump
events=gpf+abort
filter=
label=/home/user/Snap%seq.%Y%m%d.%H%M%S.%pid.%seq.trc
range=1..0
priority=500
request=serial
opts=
```

```
-----
```

```

dumpFn=doSnapDump
events=systhrow
filter=java/lang/OutOfMemoryError
label=/home/user/Snap%seq.%Y%m%d.%H%M%S.%pid.%seq.trc
range=1..4
priority=500
request=serial
opts=-----
dumpFn=doHeapDump
events=systhrow
filter=java/lang/OutOfMemoryError
label=/home/user/heapdump.%Y%m%d.%H%M%S.%pid.%seq.phd
range=1..4
priority=40
request=exclusive+prewalk
opts=PHD
-----
dumpFn=doJavaDump
events=gpf+user+abort
filter=
label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt
range=1..0
priority=10
request=exclusive
opts=
-----
dumpFn=doJavaDump
events=systhrow
filter=java/lang/OutOfMemoryError
label=/home/user/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt
range=1..4
priority=10
request=exclusive
opts=
-----

```

Removing dump agents

You can remove all default dump agents and any preceding dump options by using **-Xdump:none**.

Use this option so that you can subsequently specify a completely new dump configuration.

You can also remove dump agents of a particular type. For example, to turn off all Heapdumps (including default agents) but leave Javacore enabled, use the following option:

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

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 439 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
 - OUTFOFMEMORY
- *<agent>* can be:
 - ALL
 - NONE
 - JAVADUMP
 - SYSDUMP
 - HEAPDUMP
 - CEEDUMP (z/OS specific)
- *<count>* is the number of times to run the specified agent for the specified condition. This value is optional. By default, the agent will run every time the condition occurs. This option is introduced in Java 5 SR9.

JAVA_DUMP_OPTS is parsed by taking the leftmost occurrence of each condition, so duplicates are ignored. The following setting will produce a system dump for the first error condition only:

```
ONERROR(SYSDUMP[1]),ONERROR(JAVADUMP)
```

Also, the **ONANYSIGNAL** condition is parsed before all others, so

```
ONINTERRUPT(NONE),ONANYSIGNAL(SYSDUMP)
```

has the same effect as

```
ONANYSIGNAL(SYSDUMP),ONINTERRUPT(NONE)
```

If the **JAVA_DUMP_TOOL** environment variable is set, that variable is assumed to specify a valid executable name and is parsed for replaceable fields, such as %pid. If %pid is detected in the string, the string is replaced with the JVM's own process ID. The tool specified by **JAVA_DUMP_TOOL** is run after any system dump or Heapdump has been taken, before anything else.

Other environments variables available for controlling dumps are listed in “Javdump and Heapdump options” on page 416.

From Java 5 SR 9, 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 237.

3. **DISABLE_JAVADUMP**, **IBM_HEAPDUMP**, and **IBM_HEAP_DUMP** environment variables.
4. **IBM_JAVADUMP_OUTOFMEMORY** and **IBM_HEAPDUMP_OUTOFMEMORY** environment variables.
5. **JAVA_DUMP_OPTS** environment variable.
6. Remaining **-Xdump** command-line options.

Prior to Java 5 SR 9, the **DISABLE_JAVADUMP**, **IBM_HEAPDUMP**, and **IBM_HEAP_DUMP** environment variables took precedence over the **JAVA_DUMP_OPTS** environment variable.

From Java 5 SR 9, setting **JAVA_DUMP_OPTS** only affects those conditions you specify. Actions on other conditions are left unchanged. Prior to Java 5 SR 9, setting **JAVA_DUMP_OPTS** overrides settings for all the conditions.

Signal mappings

The signals used in the **JAVA_DUMP_OPTS** environment variable map to multiple operating system signals.

The mapping of operating system signals to the "condition" when you are setting the **JAVA_DUMP_OPTS** environment variable is as follows:

	z/OS	Windows	Linux, AIX, and i5/OS
EXCEPTION	SIGTRAP		SIGTRAP
	SIGILL	SIGILL	SIGILL
	SIGSEGV	SIGSEGV	SIGSEGV
	SIGFPE	SIGFPE	SIGFPE
	SIGBUS		SIGBUS
	SIGSYS		
	SIGXCPU		SIGXCPU
	SIGXFSZ		SIGXFSZ
INTERRUPT	SIGINT	SIGINT	SIGINT
	SIGTERM	SIGTERM	SIGTERM
	SIGHUP		SIGHUP
ERROR	SIGABRT	SIGABRT	SIGABRT
DUMP	SIGQUIT		SIGQUIT
		SIGBREAK	

Windows, Linux, AIX, and i5/OS 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.txt`. See Chapter 22, "Using Javacore," on page 245 for more information.
- **Heapdumps:** Output is written to a file named `heapdump.%Y%m%d.%H%M%S.%pid.phd`. See Chapter 23, "Using Heapdump," on page 257 for more information.

System dumps on Linux

Linux does not provide an operating system API for generating a system dump from a running process. The JVM produces system dumps on Linux by using the `fork()` API to start an identical process to the parent JVM process. The JVM then generates a `SIGSEGV` signal in the child process. The `SIGSEGV` signal causes Linux to create a system dump for the child process. The parent JVM processes and renames the system dump, as required, by the **-Xdump** options, and might add additional data into the dump file.

The system dump for the child process contains an exact copy of the memory areas used in the parent. The SDK dump viewer can obtain information about the Java threads, classes, and heap from the system dump. However, the dump viewer, and other system dump debuggers show only the single native thread that was running in the child process.

The Linux `kernel.core_pattern` setting (available in Linux 2.5 and later kernels) can be used to specify the name and path for system dumps. The JVM dump agents override the Linux system dump name and path by renaming the dump as specified in the **-Xdump** options. If the `kernel.core_pattern` setting specifies a different file system to the **-Xdump** options, the JVM dump agents might be unable to change the file path. In this case the JVM renames the dump, but leaves the file path unchanged. You can find the dump file name and location in the `JVMDUMP010I` message.

z/OS specifics

Dump output is written to different files, depending on the type of the dump. File names include a time stamp. The z/OS platform has an additional dump type called `CEEDUMP`.

From Java 5 SR9, the `CEEDUMP` is not produced by default. Use the `ceedump` dump agent to enable `CEEDUMP` production.

If **CEEDUMP** is specified, an LE `CEEDUMP` is produced for the relevant conditions, after any system dump processing, but before a `Javadump` is produced. A `CEEDUMP` is a formatted summary system dump that shows stack traces for each thread that is in the JVM process, together with register information and a short dump of storage for each register.

On z/OS, you can change the behavior of LE by setting the `_CEE_RUNOPTS` environment variable. See the *LE Programming Reference* for more information. In particular, the **TRAP** option determines whether LE condition handling is enabled, which, in turn, drives JVM signal handling, and the **TERMTHDACT** option indicates the level of diagnostic information that LE should produce.

For more information about **CEEDUMP** see “LE `CEEDUMPs`” on page 229

Dump filenames and locations

Dump files produced on z/OS include:

- **SYSDUMP**: On TSO as a standard MVS data set, using the default name of the form: `%uid.JVM.TDUMP.%job.D%Y%m%d.T%H%M%S`, or as determined by the setting of the `JAVA_DUMP_TDUMP_PATTERN` environment variable.

- **CEEDUMP:** In the directory specified by `_CEE_DMPTARG`, or the current directory if `_CEE_DMPTARG` is not specified, using the file name: `CEEDUMP.%Y%m%d.%H%M%S.%pid`.
- **HEAPDUMP:** In the current directory as a file named `heapdump.%Y%m%d.T%H%M%S.phd`. See Chapter 23, “Using Heapdump,” on page 257 for more information.
- **JAVADUMP:** In the same directory as `CEEDUMP`, or standard `JAVADUMP` directory as: `javacore.%Y%m%d.%H%M%S.%pid.txt`.

Default dump options

The default dump options on z/OS are different to the default dump options on other platforms. 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
-----
dumpFn=doSystemDump
events=gpf+user+abort
filter=
label=%uid.JVM.TDUMP.%job.D%y%m%d.T%H%M%S
range=1..0
priority=999
request=serial
opts=IEATDUMP
-----
dumpFn=doSnapDump
events=gpf+abort
filter=
label=/u/chamber/build/Snap.%Y%m%d.%H%M%S.%pid.%seq.trc
range=1..0
priority=500
request=serial
opts=
-----
dumpFn=doSnapDump
events=systhrow
filter=java/lang/OutOfMemoryError
label=/u/chamber/build/Snap.%Y%m%d.%H%M%S.%pid.%seq.trc
range=1..4
priority=500
request=serial
opts=
-----
dumpFn=doHeapDump
events=systhrow
filter=java/lang/OutOfMemoryError
label=/u/chamber/build/heapdump.%Y%m%d.%H%M%S.%pid.%seq.phd
range=1..4
priority=40
request=exclusive+compact+prewalk
opts=PHD
-----
dumpFn=doJavaDump
events=gpf+user+abort
filter=
label=/u/chamber/build/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt
range=1..0
priority=10
request=exclusive
opts=
-----
```

```
dumpFn=doJavaDump
events=systhrow
filter=java/lang/OutOfMemoryError
label=/u/chamber/build/javacore.%Y%m%d.%H%M%S.%pid.%seq.txt
range=1..4
priority=10
request=exclusive
opts=
-----
```

Chapter 22. Using Jvadump

Jvadump produces files that contain diagnostic information related to the JVM and a Java application captured at a point during execution. For example, the information can be about the operating system, the application environment, threads, stacks, locks, and memory.

The exact contents depend on the platform on which you are running. By default, a Jvadump occurs when the JVM terminates unexpectedly. A Jvadump can also be triggered by sending specific signals to the JVM. Jvadumps are human readable.

The preferred way to control the production of Jvadumps is by enabling dump agents (see Chapter 21, “Using dump agents,” on page 223) using **-Xdump:java:** on application startup. You can also control Jvadumps by the use of environment variables. See “Environment variables and Jvadump” on page 256.

Default agents are in place that (if not overridden) create Jvadumps when the JVM terminates unexpectedly or when an out-of-memory exception occurs. Jvadumps are also triggered by default when specific signals are received by the JVM.

Note: **Jvadump** is also known as **Javacore**. Javacore is NOT the same as a **core file**, which is generated by a system dump.

This chapter describes:

- “Enabling a Jvadump”
- “Triggering a Jvadump”
- “Interpreting a Jvadump” on page 247
- “Environment variables and Jvadump” on page 256

Enabling a Jvadump

Jvadumps are enabled by default. You can turn off the production of Jvadumps with **-Xdump:java:none**.

You are not recommended to turn off Jvadumps because they are an essential diagnostics tool.

Use the **-Xdump:java** option to give more fine-grained control over the production of Jvadumps. See Chapter 21, “Using dump agents,” on page 223 for more information.

Triggering a Jvadump

Jvadumps can be triggered by error conditions, or can be initiated in a number of ways to obtain diagnostic information.

Jvadumps triggered by error conditions

By default, a Jvadump is triggered when one of the following error conditions occurs:

A fatal native exception

Not a Java Exception. A “fatal” exception is one that causes the JVM to stop. The JVM handles the event by producing a system dump followed by a snap trace file, a Javadump, and then terminating the process.

The JVM has insufficient memory to continue operation

There are many reasons for running out of memory. See Part 3, “Problem determination,” on page 83 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 Linux 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 signal for z/OS is SIGQUIT. Use the command `kill -QUIT n` to send the signal to a process with process id (PID) `n`. Alternatively, press **CTRL+V** in the shell window that started Java.

The 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.

On Windows systems, use the keyboard combination **CTRL+Break** in the command window that started Java to trigger the Javadump.

The signal for i5/OS is SIGQUIT. Use the command `kill -QUIT n` to send the signal to a process with process id (PID) `n`. The PID for a particular JVM can be found in the joblog for the job (JVAB56D: Java Virtual Machine is IBM Technology for Java. PID(x)), or using the “ps” command from qsh or an i5/OS PASE shell.

The JVM continues after the signal has been handled.

You can use the `JavaDump()` method in your application

The `com.ibm.jvm.Dump` class contains a static `JavaDump()` method that causes Java code to initiate a Javadump. In your application code, add a call to `com.ibm.jvm.Dump.JavaDump()`. This call is subject to the same Javadump environment variables that are described in “Enabling a Javadump” on page 245.

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 223 for more information.

You can use the `trigger` trace option to generate a Javadump

Use the `-Xtrace:trigger` option to produce a Javadump by calling the `substring` method shown in the following example:

```
-Xtrace:trigger=method{java/lang/String.substring,javadump}
```

For a detailed description of this trace option, see
“trigger=<clause>[,<clause>][,<clause>]...” on page 301

Interpreting a Jav_dump

This section gives examples of the information contained in a Jav_dump and how it can be useful in problem solving.

The content and range of information in a Jav_dump might change between JVM versions or service refreshes. Some information might be missing, depending on the operating system platform and the nature of the event that produced the Jav_dump.

Jav_dump tags

The Jav_dump file contains sections separated by eyecatcher title areas to aid readability of the Jav_dump.

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.

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

```
1CIJAVAVERSION J2RE 5.0 IBM J9 2.3 Windows XP x86-32 build 20051012_03606_1HdSMR
(JIT enabled - 20051012_1800_r8)
```

Normal tags have these characteristics:

- Tags are up to 15 characters long (padded with spaces).
- The first digit is a nesting level (0,1,2,3).
- The second and third characters identify the section of the dump. The major sections are:
 - CI** Command-line interpreter
 - CL** Class loader
 - LK** Locking
 - ST** Storage (Memory management)
 - TI** Title
 - XE** Execution engine
- The remainder is a unique string, JAVAVERSION in the previous example.

Special tags have these characteristics:

- A tag of NULL means the line is just to aid readability.
- Every section is headed by a tag of 0SECTION with the section title.

Here is an example of some tags taken from the start of a dump. The components are highlighted for clarification.

Windows:

```
NULL -----
0SECTION  TITLE subcomponent dump routine
NULL =====
1TISIGINFO Dump Event "gpf" (00002000) received
1TIDATETIME Date: 2008/10/22 at 12:56:49
```

```

1TIFILENAME    Javacore filename: /home/javacore.20081022.125648.2014.0003.txt
NULL          -----
0SECTION      GPINFO subcomponent dump routine
NULL          =====
2XHOSLEVEL    OS Level : Linux 2.6.14-ray8.1smp
2XHCPUS       Processors -
3XHCPUARCH    Architecture : x86
3XHNUMCPUS    How Many : 1

```

Other platforms:

```

NULL          -----
0SECTION      TITLE subcomponent dump routine
NULL          =====
1TISIGINFO    Dump Event "user" (00004000) received
1TIDATETIME   Date: 2009/06/03 at 06:54:19
1TIFILENAME    Javacore filename: /home/user/javacore.20090603.065419.315480.0001.txt
NULL          -----
0SECTION      GPINFO subcomponent dump routine
NULL          =====
2XHOSLEVEL    OS Level : AIX 6.1
2XHCPUS       Processors -
3XHCPUARCH    Architecture : ppc
3XHNUMCPUS    How Many : 8
3XHNUMASUP    NUMA is either not supported or has been disabled by user

```

For the rest of the topics in this section, the tags are removed to aid readability.

TITLE, GPINFO, and ENVINFO sections

At the start of a Javadump, the first three sections are the TITLE, GPINFO, and ENVINFO sections. They provide useful information about the cause of the dump.

The following example shows some output taken from a simple Java test program calling (using JNI) an external function that causes a “general protection fault” (GPF).

TITLE

Shows basic information about the event that caused the generation of the Javadump, the time it was taken, and its name.

GPINFO

Varies in content depending on whether the Javadump was produced because of a GPF or not. It shows some general information about the operating system. If the failure was caused by a GPF, GPF information about the failure is provided, in this case showing that the protection exception was thrown from MVSCR71D.dll. The registers specific to the processor and architecture are also displayed.

The GPINFO section also refers to the vmState, recorded in the console output as VM flags. The vmState is the thread-specific state of what was happening in the JVM at the time of the crash. The value for vmState is a 32-bit hexadecimal number of the format MMMMSSSS, where MMMM is the major component and SSSS is component specific code.

Major component	Code number
INTERPRETER	0x10000
GC	0x20000
GROW_STACK	0x30000
JNI	0x40000

Major component	Code number
JIT_CODEGEN	0x50000
BCVERIFY	0x60000
RTVERIFY	0x70000
SHAREDCLASSES	0x80000

In the following example, the value for vmState is VM flags:00040000, which indicates a crash in the JNI component.

When the vmState major component is JNI, the crash might be caused by customer JNI code or by Java SDK JNI code. Check the Javadump to reveal which JNI routine was called at the point of failure. The JNI is the only component where a crash might be caused by customer code.

When the vmState major component is JIT_CODEGEN, see the information at Chapter 26, “JIT problem determination,” on page 317.

ENVINFO

Shows information about the JRE level that failed and details about the command line that launched the JVM process and the JVM environment in place.

```

-----
TITLE subcomponent dump routine
=====
Dump Event "gpf" (00002000) received
Date:                2005/10/24 at 11:08:59
Javacore filename:   C:\Program Files\IBM\Java50\jre\bin\javacore.20051024.110853.2920.txt

-----
GPINFO subcomponent dump routine
=====
OS Level           : Windows XP 5.1 build 2600 Service Pack 1
Processors -
    Architecture   : x86
    How Many       : 1

J9Generic_Signal_Number: 00000004
ExceptionCode: C0000005
ExceptionAddress: 423155F1
ContextFlags: 0001003F
Handler1: 70C2FE60
Handler2: 70B86AB0
InaccessibleAddress: 000004D2

Module: C:\WINDOWS\System32\MSVCR71D.dll
Module_base_address: 42300000
Offset_in_DLL: 000155F1

Registers:
    EDI:000004D2
    ESI:00000020
    EAX:000004D2
    EBX:00000000
    ECX:000004D2
    EDX:00000000
    EIP:423155F1
    ESP:0007FBF4
    EBP:0007FCDC

VM flags:00040000
-----

```

```

ENVINFO subcomponent dump routine
=====
J2RE 5.0 IBM J9 2.3 Windows XP x86-32 build 20051015_03657_1HdSMR (JIT enabled - 20051015_1812_r8)
Running as a standalone JVM
java Test GPF
Java Home Dir: C:\Program Files\IBM\Java50\jre
Java DLL Dir: C:\Program Files\IBM\Java50\jre\bin
Sys Classpath: C:\Program Files\IBM\Java50\jre\lib\vm.jar;C:\Program Files\.....
UserArgs:
-Xjcl:jclscar_23
-Dcom.ibm.oti.vm.bootstrap.library.path=C:\Program Files\IBM\Java50\jre\bin
-Dsun.boot.library.path=C:\Program Files\IBM\Java50\jre\bin
    << lines removed .....>>
-Xdump

```

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

```

Module: C:\WINDOWS\System32\MSVCR71D.dll
Module_base_address: 42300000
Offset_in_DLL: 000155F1

```

You can see that a crash occurred in MSVCR71D.dll, that was loaded at 42300000, and the crash point was at offset 0x155F1 in MSVCR71D.dll.

From Java 5 SR11 onwards, 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 environment variables that are in force.
- The system ulimits, or user limits, in place. These values are shown only on UNIX platforms.

The output is similar to the following lines:

```

User Limits (in bytes except for NOFILE and NPROC)
-----
type soft limit hard limit
RLIMIT_AS unlimited unlimited
RLIMIT_CORE 0 unlimited
RLIMIT_CPU unlimited unlimited
RLIMIT_DATA unlimited unlimited
RLIMIT_FSIZE unlimited unlimited
RLIMIT_LOCKS unlimited unlimited
RLIMIT_MEMLOCK 32768 32768
....

Environment Variables
-----
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=::ffff:9.20.184.180 1655 22
OLDPWD=/home/test
SSH_TTY=/dev/pts/1
USER=test
MAIL=/var/spool/mail/test
PATH=/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/test/bin
LANG=en_GB.UTF-8

```

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 Chapter 2, “Memory management,” on page 7 for details about how the Memory Manager works.

This part of the Javadump gives various storage management values (in hexadecimal), including the free space and current size of the heap. It also contains garbage collection history data, described in “Default memory management tracing” on page 285. Garbage collection history data is shown as a sequence of tracepoints, each with a timestamp, ordered with the most recent tracepoint first.

In the Javadump, segments are blocks of memory allocated by the Java runtime for tasks that use large amounts of memory. Example tasks are maintaining JIT caches, and storing Java classes. The Java runtime also allocates other native memory, that is not listed in the MEMINFO section. The total memory used by Java runtime segments does not necessarily represent the complete memory footprint of the Java runtime. A Java runtime segment consist of the segment data structure, and an associated block of native memory.

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

Alloc The address of the top of the section of associated native memory that is currently in use. For some segment types, this address is the same as the end address.

Bytes The size of the attached native memory.

End The end address of the attached storage.

Segment
The address of the segment data structure.

Start The start address of the associated native memory.

Type The internal bit-field describing the characteristics of the associated native memory.

```
-----
MEMINFO subcomponent dump routine
=====
Bytes of Heap Space Free: 365df8
Bytes of Heap Space Allocated: 400000
```

Internal Memory						
segment	start	alloc	end	type	bytes	
00172FB8	41D79078	41D7DBC4	41D89078	01000040	10000	
<< lines removed for clarity >>						
00172ED4	4148C368	4149C360	4149C368	01000040	10000	
Object Memory						
segment	start	alloc	end	type	bytes	
00173EDC	00420000	00820000	00820000	00000009	400000	
Class Memory						
segment	start	alloc	end	type	bytes	
001754C8	41E36250	41E36660	41E3E250	00010040	8004	
<< lines removed for clarity >>						
00174F24	41531C70	415517C8	41551C70	00020040	20000	
JIT Code Cache						
segment	start	alloc	end	type	bytes	
4148836C	002F0000	00370000	00370000	00000068	80000	
JIT Data Cache						
segment	start	alloc	end	type	bytes	
41489374	416A0020	416A259C	41720020	00000048	80000	

```
GC History
10:11:18:562797000 GMT j9mm.53 - GlobalGC end: workstackoverflow=0 overflowcount=0 weakrefs=81
    soft=1 phantom=0 finalizers=21 newspace=0/0 oldspace=64535568/101534208 loa=6582784/9938432
10:11:18:562756000 GMT j9mm.57 - Sweep end
10:11:18:561695000 GMT j9mm.56 - Sweep start
```

```

10:11:18:561692000 GMT j9mm.55 - Mark end
10:11:18:558022000 GMT j9mm.54 - Mark start
10:11:18:558003000 GMT j9mm.52 - GlobalGC start: weakrefs=81 soft=1 phantom=0 finalizers=21
    globalcount=41 scavengecount=0
10:11:18:465931000 GMT j9mm.53 - GlobalGC end: workstackoverflow=0 overflowcount=0 weakrefs=81
    soft=1 phantom=0 finalizers=21 newspace=0/0 oldspace=52177296/103607808 loa=4944968/9105408
10:11:18:465892000 GMT j9mm.57 - Sweep end
10:11:18:464689000 GMT j9mm.56 - Sweep start
10:11:18:464687000 GMT j9mm.55 - Mark end
10:11:18:460946000 GMT j9mm.54 - Mark start
10:11:18:460927000 GMT j9mm.52 - GlobalGC start: weakrefs=83 soft=1 phantom=0 finalizers=21
    globalcount=40 scavengecount=0
10:11:18:350282000 GMT j9mm.53 - GlobalGC end: workstackoverflow=0 overflowcount=0 weakrefs=83
    soft=1 phantom=0 finalizers=21 newspace=0/0 oldspace=53218040/104210432 loa=1838432/8116224
10:11:18:350263000 GMT j9mm.57 - Sweep end
10:11:18:349051000 GMT j9mm.56 - Sweep start
10:11:18:349048000 GMT j9mm.55 - Mark end
10:11:18:345270000 GMT j9mm.54 - Mark start

```

Locks, monitors, and deadlocks (LOCKS)

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

A lock, also referred to as a monitor, prevents more than one entity from accessing a shared resource. Each object in Java has an associated lock, obtained by using a synchronized block or method. In the case of the JVM, threads compete for various resources in the JVM and locks on Java objects.

This example was taken from a deadlock test program where two threads “DeadLockThread 0” and “DeadLockThread 1” were unsuccessfully attempting to synchronize (Java keyword) on two java/lang/Integers.

You can see in the example (highlighted) that “DeadLockThread 1” has locked the object instance java/lang/Integer@004B2290. The monitor has been created as a result of a Java code fragment looking like “synchronize(count0)”, and this monitor has “DeadLockThread 1” waiting to get a lock on this same object instance (count0 from the code fragment). Below the highlighted section is another monitor locked by “DeadLockThread 0” that has “DeadLockThread 1” waiting.

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

```

-----
LOCKS subcomponent dump routine
=====

```

```

Monitor pool info:
Current total number of monitors: 2

```

Monitor Pool Dump (flat & inflated object-monitors):

```

sys_mon_t:0x00039B40 infl_mon_t: 0x00039B80:
  java/lang/Integer@004B22A0/004B22AC: Flat locked by "DeadLockThread 1"
                                     (0x41DAB100), entry count 1
    Waiting to enter:
      "DeadLockThread 0" (0x41DAAD00) sys_mon_t:0x00039B98 infl_mon_t: 0x00039BD8:
      java/lang/Integer@004B2290/004B229C: Flat locked by "DeadLockThread 0"
                                     (0x41DAAD00), entry count 1
    Waiting to enter:
      "DeadLockThread 1" (0x41DAB100)

```

JVM System Monitor Dump (registered monitors):

```

Thread global lock (0x00034878): <unowned>
NLS hash table lock (0x00034928): <unowned>
portLibrary_j9sig_async_monitor lock (0x00034980): <unowned>

```

```
Hook Interface lock (0x000349D8): <unowned>
  < lines removed for brevity >
```

```
=====
Deadlock detected !!!
-----
```

```
Thread "DeadLockThread 1" (0x41DAB100)
  is waiting for:
    sys_mon_t:0x00039B98 infl_mon_t: 0x00039BD8:
    java/lang/Integer@004B2290/004B229C:
  which is owned by:
Thread "DeadLockThread 0" (0x41DAAD00)
  which is waiting for:
    sys_mon_t:0x00039B40 infl_mon_t: 0x00039B80:
    java/lang/Integer@004B22A0/004B22AC:
  which is owned by:
Thread "DeadLockThread 1" (0x41DAB100)
```

Deadlocks can occur when serializing multiple `java.util.Hashtables` that refer to each other in different threads at the same time. Use the system property command-line option `-DCLONE_HASHTABLE_FOR_SYNCHRONIZATION` to resolve the deadlock. For more information about this option, see “System property command-line options” on page 442.

Threads and stack trace (THREADS)

For the application programmer, one of the most useful pieces of a Java dump is the THREADS section. This section shows a list of Java threads and stack traces.

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

```
"Signal Dispatcher" TID:0x41509200, j9thread_t:0x0003659C, state:R,prio=5
  (native thread ID:5820, native priority:0, native policy:SCHED_OTHER)
  at com/ibm/misc/SignalDispatcher.waitForSignal(Native Method)
  at com/ibm/misc/SignalDispatcher.run(SignalDispatcher.java:84)
```

The properties on the first line are thread name, identifier, JVM data structure address, current state, and Java priority. The properties on the second line are the native operating system thread ID, native operating system thread priority and native operating system scheduling policy.

The Java thread priority is mapped to an operating system priority value in a platform-dependent manner. A large value for the Java thread priority means that the thread has a high priority. In other words, the thread runs more frequently than lower priority threads.

The values of state can be:

- R - Runnable - the thread is able to run when given the chance.
- CW - Condition Wait - the thread is waiting. For example, because:
 - A `sleep()` call is made
 - The thread has been blocked for I/O
 - A `wait()` method is called to wait on a monitor being notified
 - The thread is synchronizing with another thread with a `join()` call
- S – Suspended – the thread has been suspended by another thread.
- Z – Zombie – the thread has been killed.

- P – Parked – the thread has been parked by the new concurrency API (java.util.concurrent).
- B – Blocked – the thread is waiting to obtain a lock that something else currently owns.

Understanding Java thread details

Below each Java thread is a stack trace, which represents the hierarchy of Java method calls made by the thread.

The following example is taken from the same Javadump that is used in the LOCKS example. Two threads, “DeadLockThread 0” and “DeadLockThread 1”, are in blocked state. The application code path that resulted in the deadlock between “DeadLockThread 0” and “DeadLockThread 1” can clearly be seen.

There is no current thread because all the threads in the application are blocked. A user signal generated the Javadump.

```
-----
THREADS subcomponent dump routine
=====
```

```
Current Thread Details
-----
```

```
All Thread Details
-----
```

```
Full thread dump J9SE VM (J2RE 5.0 IBM J9 2.3 Linux x86-32 build 20060714_07194_1HdSMR,
native threads):
"DestroyJavaVM helper thread" TID:0x41508A00, j9thread_t:0x00035EAC, state:CW, prio=5
    (native thread ID:3924, native priority:0, native policy:SCHED_OTHER, scope:00A6D068)
"JIT Compilation Thread" TID:0x41508E00, j9thread_t:0x000360FC, state:CW, prio=11
    (native thread ID:188, native priority:11, native policy:SCHED_OTHER, scope:00A6D068)
"Signal Dispatcher" TID:0x41509200, j9thread_t:0x0003659C, state:R, prio=5
    (native thread ID:3192, native priority:0, native policy:SCHED_OTHER, scope:00A6D084)
    at com/ibm/misc/SignalDispatcher.waitForSignal(Native Method)
    at com/ibm/misc/SignalDispatcher.run(SignalDispatcher.java:84)
"DeadLockThread 0" TID:0x41DAAD00, j9thread_t:0x42238A1C, state:B, prio=5
    (native thread ID:1852, native priority:0, native policy:SCHED_OTHER, scope:00A6D068)
    at Test$DeadlockThread0.SyncMethod(Test.java:112)
    at Test$DeadlockThread0.run(Test.java:131)
"DeadLockThread 1" TID:0x41DAB100, j9thread_t:0x42238C6C, state:B, prio=5
    (native thread ID:1848, native priority:0, native policy:SCHED_OTHER, scope:00A6D068)
    at Test$DeadlockThread1.SyncMethod(Test.java:160)
    at Test$DeadlockThread1.run(Test.java:141)
```

Current Thread Details section

If the Javadump is triggered on a running Java thread, the Current Thread Details section shows a Java thread name, properties and stack trace. This output is generated if, for example, a GPF occurs on a Java thread, or if the com.ibm.jvm.Dump.JavaDump() API is called.

```
Current Thread Details
-----
```

```
"main" TID:0x0018D000, j9thread_t:0x002954CC, state:R, prio=5
    (native thread ID:0xAD0, native priority:0x5, native policy:UNKNOWN)
    at com/ibm/jvm/Dump.JavaDumpImpl(Native Method)
    at com/ibm/jvm/Dump.JavaDump(Dump.java:20)
    at Test.main(Test.java:26)
```

Typically, Javadumps triggered by a user signal do not show a current thread because the signal is handled on a native thread, and the Java threads are

suspended while the Javdump is produced.

Classloaders and Classes (CLASSES)

An example of the classloader (CLASSES) section that includes Classloader summaries and Classloader loaded classes. Classloader summaries are the defined class loaders and the relationship between them. Classloader loaded classes are the classes that are loaded by each classloader.

See Chapter 3, “Class loading,” on page 29 for information about the parent-delegation model.

In this example, there are the standard three classloaders:

- Application classloader (sun/misc/Launcher\$AppClassLoader), which is a child of the extension classloader.
- The Extension classloader (sun/misc/Launcher\$ExtClassLoader), which is a child of the bootstrap classloader.
- The Bootstrap classloader. Also known as the System classloader.

The example that follows shows this relationship. Take the application classloader with the full name sun/misc/Launcher\$AppClassLoader. Under Classloader summaries, it has flags -----ta-, which show that the class loader is t=trusted and a=application (See the example for information on class loader flags). It gives the number of loaded classes (1) and the parent classloader as sun/misc/Launcher\$ExtClassLoader.

Under the ClassLoader loaded classes heading, you can see that the application classloader has loaded three classes, one called Test at address 0x41E6CFE0.

In this example, the System class loader has loaded a large number of classes, which provide the basic set from which all applications derive.

```
-----
CLASSES subcomponent dump routine
=====
Classloader summaries
  12345678: 1=primordial,2=extension,3=shareable,4=middleware,
            5=system,6=trusted,7=application,8=delegating
  p---st--   Loader *System*(0x00439130)
            Number of loaded classes 306
  -x--st--   Loader sun/misc/Launcher$ExtClassLoader(0x004799E8),
            Parent *none*(0x00000000)
            Number of loaded classes 0
  -----ta- Loader sun/misc/Launcher$AppClassLoader(0x00484AD8),
            Parent sun/misc/Launcher$ExtClassLoader(0x004799E8)
            Number of loaded classes 1

Classloader loaded classes
Loader *System*(0x00439130)
  java/security/CodeSource(0x41DA00A8)
  java/security/PermissionCollection(0x41DA0690)
  << 301 classes removed for clarity >>
  java/util/AbstractMap(0x4155A8C0)
  java/io/OutputStream(0x4155ACB8)
  java/io/FilterOutputStream(0x4155AE70)
Loader sun/misc/Launcher$ExtClassLoader(0x004799E8)
Loader sun/misc/Launcher$AppClassLoader(0x00484AD8)
  Test(0x41E6CFE0)
  Test$DeadlockThread0(0x41E6D410)
  Test$DeadlockThread1(0x41E6D6E0)
```

Environment variables and Javadump

Although the preferred mechanism of controlling the production of Javadumps is now by the use of dump agents using **-Xdump:java**, you can also use the previous mechanism, environment variables.

The following table details environment variables specifically concerned with Javadump production:

Environment Variable	Usage Information
DISABLE_JAVADUMP	Setting DISABLE_JAVADUMP to true is the equivalent of using -Xdump:java:none and stops the default production of javadumps.
IBM_JAVACOREDIR	The default location into which the Javacore will be written.
JAVA_DUMP_OPTS	Use this environment variable to control the conditions under which Javadumps (and other dumps) are produced. See "Dump agent environment variables" on page 239 for more information.
IBM_JAVADUMP_OUTOFMEMORY	By setting this environment variable to false, you disable Javadumps for an out-of-memory exception.

Chapter 23. Using Heapdump

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

This dump is stored in a Portable Heap Dump (PHD) file, a compressed binary format. You can use various tools on the Heapdump output to analyze the composition of the objects on the heap and (for example) help to find the objects that are controlling large amounts of memory on the Java heap and the reason why the Garbage Collector cannot collect them.

This chapter describes:

- “Information for users of previous releases of Heapdump”
- “Getting Heapdumps”
- “Available tools for processing Heapdumps” on page 258
- “Using **-Xverbose:gc** to obtain heap information” on page 258
- “Environment variables and Heapdump” on page 258
- “Text (classic) Heapdump file format” on page 259

Information for users of previous releases of Heapdump

Heapdumps for the platforms described in this guide are different from previous releases of the IBM Virtual Machine for Java. Heapdumps are now produced in phd format and you can view them using a variety of tools.

For more information, see “Available tools for processing Heapdumps” on page 258. Before Version 1.4.2, Service Refresh 2, Heapdump phd files were produced using the jdmpview tool from a combination of full system dumps and the jextract post-processor tool. This technique is still supported and described in “Generating Heapdumps” on page 271.

Getting Heapdumps

By default, a Heapdump is produced when the Java heap is exhausted. Heapdumps can be generated in other situations by use of **-Xdump:heap**.

See Chapter 21, “Using dump agents,” on page 223 for more detailed information about generating dumps based on specific events. Heapdumps can also be generated programmatically by use of the `com.ibm.jvm.Dump.HeapDump()` method from inside the application code.

To see which events will trigger a dump, use **-Xdump:what**. See Chapter 21, “Using dump agents,” on page 223 for more information.

By default, Heapdumps are produced in PHD format. To produce Heapdumps in text format, see “Enabling text formatted (“classic”) Heapdumps” on page 258.

Environment variables can also affect the generation of Heapdumps (although this is a deprecated mechanism). See “Environment variables and Heapdump” on page 258 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 Chapter 21, “Using dump agents,” on page 223). For example:

- **-Xdump:heap:opts=CLASSIC** will start the default Heapdump agents using classic rather than PHD output.
- **-Xdump:heap:defaults:opts=CLASSIC+PHD** will enable both classic and PHD output by default for all Heapdump agents.

You can also define one of the following environment variables:

- **IBM_JAVA_HEAPDUMP_TEST**, which allows you to perform the equivalent of **opts=PHD+CLASSIC**
- **IBM_JAVA_HEAPDUMP_TEXT**, which allows the equivalent of **opts=CLASSIC**

Available tools for processing Heapdumps

There are several tools available for Heapdump analysis through IBM support Web sites.

The preferred Heapdump analysis tool is the IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer. The tool is available in IBM Support Assistant: <http://www.ibm.com/software/support/isa/>. Information about the tool can be found at <http://www.ibm.com/developerworks/java/jdk/tools/memoryanalyzer/>

Further details of the range of available tools can be found at <http://www.ibm.com/support/docview.wss?uid=swg24009436>

Using -Xverbose:gc to obtain heap information

Use the **-Xverbose:gc** utility to obtain information about the Java Object heap in real time while running your Java applications.

To activate this utility, run Java with the **-verbose:gc** option:

```
java -verbose:gc
```

For more information, see Chapter 2, “Memory management,” on page 7.

Environment variables and Heapdump

Although the preferred mechanism for controlling the production of Heapdumps is now the use of dump agents with **-Xdump:heap**, you can also use the previous mechanism, environment variables.

The following table details environment variables specifically concerned with Heapdump production:

Environment Variable	Usage Information
IBM_HEAPDUMP IBM_HEAP_DUMP	Setting either of these to any value (such as true) enables heap dump production by means of signals.
IBM_HEAPDUMPDIR	The default location into which the Heapdump will be written.
JAVA_DUMP_OPTS	Use this environment variable to control the conditions under which Heapdumps (and other dumps) are produced. See “Dump agent environment variables” on page 239 for more information .
IBM_HEAPDUMP_OUTOFMEMORY	By setting this environment variable to false, you disable Heapdumps for an OutOfMemory condition.
IBM_JAVA_HEAPDUMP_TEST	Use this environment variable to cause the JVM to generate both phd and text versions of Heapdumps. Equivalent to opts=PHD+CLASSIC on the -Xdump:heap option.
IBM_JAVA_HEAPDUMP_TEXT	Use this environment variable to cause the JVM to generate a text (human readable) Heapdump. Equivalent to opts=CLASSIC on the -Xdump:heap option.

Text (classic) Heapdump file format

The text or classic Heapdump is a list of all object instances in the heap, including object type, size, and references between objects. On z/OS, the Heapdump is in EBCDIC.

Header record

The header record is a single record containing a string of version information.

// Version: <version string containing SDK level, platform and JVM build level>

Example:

// Version: J2RE 5.0 IBM J9 2.3 Windows XP x86-32 build 20060915_08260_1HdSMR

Object records

Object records are multiple records, one for each object instance on the heap, providing object address, size, type, and references from the object.

*<object address, in hexadecimal> [<length in bytes of object instance, in decimal>
OBJ <object type> <class block reference, in hexadecimal>
<heap reference, in hexadecimal <heap reference, in hexadecimal> ...*

The object address and heap references are in the heap, but the class block address is outside the heap. All references found in the object instance are listed, including references that are null values. The object type is either a class name including package or a primitive array or class array type, shown by its standard JVM type signature, see “Java VM type signatures” on page 261. Object records can also contain additional class block references, typically in the case of reflection class instances.

Examples:

An object instance, length 28 bytes, of type java/lang/String:

```
0x00436E90 [28] OBJ java/lang/String
```

A class block address of java/lang/String, followed by a reference to a char array instance:

```
0x415319D8 0x00436EB0
```

An object instance, length 44 bytes, of type char array:

```
0x00436EB0 [44] OBJ [C
```

A class block address of char array:

```
0x41530F20
```

An object of type array of java/util/Hashtable Entry inner class:

```
0x004380C0 [108] OBJ [Ljava/util/Hashtable$Entry;
```

An object of type java/util/Hashtable Entry inner class:

```
0x4158CD80 0x00000000 0x00000000 0x00000000 0x00000000 0x00421660 0x004381C0  
0x00438130 0x00438160 0x00421618 0x00421690 0x00000000 0x00000000 0x00000000  
0x00438178 0x004381A8 0x004381F0 0x00000000 0x004381D8 0x00000000 0x00438190  
0x00000000 0x004216A8 0x00000000 0x00438130 [24] OBJ java/util/Hashtable$Entry
```

A class block address and heap references, including null references:

```
0x4158CB88 0x004219B8 0x004341F0 0x00000000
```

Class records

Class records are multiple records, one for each loaded class, providing class block address, size, type, and references from the class.

```
<class block address, in hexadecimal> [<length in bytes of class block, in decimal>]  
CLS <class type>  
<class block reference, in hexadecimal> <class block reference, in hexadecimal> ...  
<heap reference, in hexadecimal> <heap reference, in hexadecimal>...
```

The class block address and class block references are outside the heap, but the class record can also contain references into the heap, typically for static class data members. All references found in the class block are listed, including those that are null values. The class type is either a class name including package or a primitive array or class array type, shown by its standard JVM type signature, see “Java VM type signatures” on page 261.

Examples:

A class block, length 168 bytes, for class java/lang/Runnable:

```
0x41532E68 [168] CLS java/lang/Runnable
```

References to other class blocks and heap references, including null references:

```
0x4152F018 0x41532E68 0x00000000 0x00000000 0x00499790
```

A class block, length 168 bytes, for class java/lang/Math:

```
0x00000000 0x004206A8 0x00420720 0x00420740 0x00420760 0x00420780 0x004207B0  
0x00421208 0x00421270 0x00421290 0x004212B0 0x004213C8 0x00421458 0x00421478  
0x00000000 0x41589DE0 0x00000000 0x4158B340 0x00000000 0x00000000 0x00000000  
0x4158ACE8 0x00000000 0x4152F018 0x00000000 0x00000000 0x00000000
```

Trailer record 1

Trailer record 1 is a single record containing record counts.

```
// Breakdown - Classes: <class record count, in decimal>,  
Objects: <object record count, in decimal>,  
ObjectArrays: <object array record count, in decimal>,  
PrimitiveArrays: <primitive array record count, in decimal>
```

Example:

```
// Breakdown - Classes: 321, Objects: 3718, ObjectArrays: 169,  
PrimitiveArrays: 2141
```

Trailer record 2

Trailer record 2 is a single record containing totals.

```
// EOF: Total 'Objects',Refs(null) :  
<total object count, in decimal>,  
<total reference count, in decimal>  
(,total null reference count, in decimal>)
```

Example:

```
// EOF: Total 'Objects',Refs(null) : 6349,23240(7282)
```

Java VM type signatures

The Java VM type signatures are abbreviations of the Java types are shown in the following table:

Java VM type signatures	Java type
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L <fully qualified-class> ;	<fully qualified-class>
[<type>	<type>[] (array of <type>)
(<arg-types>) <ret-type>	method

Chapter 24. Using system dumps and the dump viewer

The JVM can generate native system dumps, also known as core dumps, under configurable conditions. System dumps are typically quite large. Most tools used to analyze system dumps are also platform-specific; for example, windbg on Windows and gdb on Linux.

Dump agents are the primary method for controlling the generation of system dumps. See Chapter 21, “Using dump agents,” on page 223 for more information. To maintain backwards compatibility, the JVM supports the use of environment variables for system dump triggering. See “Dump agent environment variables” on page 239 for more information.

The dump viewer is a cross-platform tool to analyze system dumps at the JVM level rather than at the platform level.

This chapter tells you about system dumps and how to use the dump viewer. It contains these topics:

- “Overview of system dumps”
- “System dump defaults” on page 264
- “Overview of the dump viewer” on page 264
- “Dump viewer commands” on page 267
- “Example session” on page 273

Overview of system dumps

The JVM can produce system dumps in response to specific events. A system dump is a raw binary dump of the process memory when the dump agent is triggered by a failure or by an event for which a dump is requested.

Generally, you use a tool to examine the contents of a system dump. A dump viewer tool is provided in the SDK, as described in this section, or you could use a platform-specific debugger, such as windbg on Windows, gdb on Linux, or dbx on AIX, to examine the dump.

For dumps triggered by a General Protection Fault (GPF), dumps produced by the JVM contain some context information that you can read. You can find this failure context information by searching in the dump for the eye-catcher

J9Generic_Signal_Number

For example:

```
J9Generic_Signal_Number=00000004 ExceptionCode=c0000005 ExceptionAddress=7FAB506D ContextFlags=00000000  
Handler1=7FEF79C0 Handler2=7FED8CF0 InaccessibleAddress=0000001C  
EDI=41FEC3F0 ESI=00000000 EAX=41FB0E60 EBX=41EE6C01  
ECX=41C5F9C0 EDX=41FB0E60  
EIP=7FAB506D ESP=41C5F948 EBP=41EE6CA4  
Module=E:\testjava\jdk\jre\bin\j9jit23.dll  
Module_base_address=7F8D0000 Offset_in_DLL=001e506d
```

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

Dump agents are the primary method for controlling the generation of system dumps. See Chapter 21, “Using dump agents,” on page 223 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:

```
dumpFn=doSystemDump
events=gpf+abort
filter=
label=/home/user/tests/core.%Y%m%d.%H%M%S.%pid.dmp
range=1..0
priority=999
request=serial
opts=
```

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.

Overview of 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 when the dump was initiated. The SDK dump viewer enables you to navigate around the dump, and obtain information in a readable form, including 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 z/OS dumps on a Windows platform.

Using the dump extractor, jextract

To use the full facilities of the dump viewer, you must first run the `jextract` tool on the system dump. The `jextract` tool obtains platform-specific information such as word size, endianness, data structure layouts, and symbolic information. It puts this information into an XML file. You must run the `jextract` tool on the same platform and the same JVM level that was being used when the dump was produced. The combination of the dump file and the XML file produced by `jextract` enables the dump viewer (`jdmpview`) to analyze and display Java information.

The extent to which `jextract` can analyze the information in a dump is affected by the state of the JVM when it was taken. For example, the dump might have been taken while the JVM was in an inconsistent state. The `exclusive` and `prewalk` dump options ensure that the JVM (and the Java heap) is in a safe state before taking a system dump:

```
-Xdump:system:defaults:request=exclusive+prewalk
```


Setting this option adds a significant overhead to taking a system dump; this overhead could cause problems in rare situations. This option is not enabled by default.

jextract is in the directory `sdk/jre/bin`.

To invoke jextract, at a command prompt type:

```
jextract <dumpfile> [<outputfile>]
```

On z/OS, you can copy the dump to an HFS file and pass that as input to jextract. Alternatively, you can pass a fully qualified MVS data set name as input to jextract. jextract is unable to read data sets larger than 2 GB directly using a 31-bit JVM. You must use COPYDUMP first or move the dump to HFS. An example of the jextract command is:

```
> jextract USER1.JVM.TDUMP.SSHD6.D070430.T092211
Loading dump file...
Read memory image from USER1.JVM.TDUMP.SSHD6.D070430.T092211
VM set to 10BA5028
Dumping JExtract file to USER1.JVM.TDUMP.SSHD6.D070430.T092211.xml
<!-- extracting gpf state -->
...
Finished writing JExtract file in 5308ms
Creating zip file: USER1.JVM.TDUMP.SSHD6.D070430.T092211.zip
Adding "USER1.JVM.TDUMP.SSHD6.D070430.T092211" to zip
Adding "USER1.JVM.TDUMP.SSHD6.D070430.T092211.xml" to zip
Adding "/u/build/sdk/jre/lib/J9TraceFormat.dat" to zip
jextract complete.
```

This produces a compressed (.zip) file in the current HFS directory.

The jextract tool accepts these parameters:

-nozip

Do not compress the output data.

-help

Provides help information.

By default, output is written to a file called `<dumpfile>.zip` in the current directory. This file is a compressed file that contains:

- The dump
- XML produced from the dump, containing details of useful internal JVM information
- Other files that can help in diagnosing the dump (such as trace entry definition files)

Typically, you would send the compressed file to IBM for problem diagnosis. Submitting data with a problem report tells you how to do that.

To analyze the dump locally, extract the compressed file using **unzip -d dir <file>** or **jar xvf <file>**. You are also advised to run `jdumpview` from that new folder.

If you run jextract on a JVM level that is different from the one for which the dump was produced, you see the following messages:

```
J9RAS.buildID is incorrect (found e8801ed67d21c6be, expecting eb4173107d21c673).
This version of jextract is incompatible with this dump.
Failure detected during jextract, see previous message(s).
```

You can still use the dump viewer on the dump, but the detail produced is limited.

The contents of the compressed file and the XML file produced are subject to change.

Using the dump viewer, jdmpview

The dump viewer is a cross-platform tool that you use to examine the contents of system dumps produced from the JVM. To be able to analyze platform-specific dumps, the dump viewer can use metadata created by the jextract tool. The dump viewer allows you to view both Java and operating system information from the time the dump was produced.

jdmpview is in the directory `sdk/bin`.

To start jdmpview, at a command prompt type:

```
jdmpview dumpfile
```

The jdmpview tool accepts these parameters:

- d**<*dumpfile*>
Specify a dump file.
- w**<*workingdir*>
Specify a writable directory.
- o**<*outputfile*>
Specify an output file.
- i**<*inputfile*>
Specify an input command file.

Typical usage is `jdmpview <dumpfile>`. The jdmpview tool opens and verifies the dump file and the associated xml file, *dumpfile.xml*.

After jdmpview processes the arguments with which it was launched, it displays the message `Ready...` When you see this message, you can start calling commands on jdmpview. You can run an unlimited number of jdmpview sessions at the same time.

You can significantly improve the performance of jdmpview against larger dumps by ensuring that your system has enough memory available to avoid paging. On larger dumps (that is, ones with large numbers of objects on the heap), you might need to start jdmpview using the **-Xmx** option to increase the maximum heap available to jdmpview:

```
jdmpview -J-Xmx<n>
```

To pass command-line arguments to the JVM, you must prefix them with **-J**. For more information about using **-Xmx**, see Appendix D, "Command-line options," on page 439.

The xml file produced by jextract on z/OS is ASCII, so that it is easily portable to other platforms for use in jdmpview and other tools. On z/OS, jdmpview expects the file to be ASCII. If you convert the file to EBCDIC and supply the converted file as input to jdmpview, you see the error messages:

```
Parsing of xml started for file CHAMBER.JVM.TDUMP.SSHD9.D070824.T094404.xml... be patient  
*** Error Message: Fatal error encountered processing incoming xml.
```

It might be useful to convert the xml file to EBCDIC for viewing on z/OS, but make sure you keep the ASCII version for use in jdmpview.

Dump viewer commands

This section describes the commands available in jdmpview. Many of the commands have short forms. For example, `display`, `dis`, and `d` are all considered equivalent in the standard command syntax. The commands are split into common subareas.

General commands

- **quit**

Short form: q

Availability: always

Terminates the jdmpview session.

- **cmds**

Availability: always

Displays the available commands at any point during the jdmpview session and also indicates which class provides the support for that command. The range of available commands might change during the session; for example, the `DIS OS` command is unavailable until after a dump has been identified.

- **help** and **help <command>**

Short form: h

Availability: always

The help command with no parameters shows general help. With a parameter, the command shows specific help on a command. For example, `help dis os` produces help information regarding the `dis os` command.

- **synonyms**

Short form: syn

Availability: always

Displays some shorthand notations recognized by the command processor. Thus, `d o 0x123456` is the equivalent of `dis obj 0x123456`.

- **display proc**

Short form: dis proc

Availability: after set dump has run

Displays information about the process or processes found in the dump. The command shows the command line that started the process, thread information, environment variables, and loaded modules.

- **display sym**

Short form: dis sym

Availability: after set dump has run

During processing, jdmpview creates symbols that enable memory addresses to be displayed as offsets from known locations (such as the start of a loaded module). This command allows the known symbols and their values to be displayed.

- **set**

Short form: s

Availability: always

Some set commands (such as `set dump`) start specific processing within `jdmpview` whereas others set and unset variables within the `jdmpview` environment.

`set` without any parameters shows which `jdmpview` variables are defined and what their values are. Similarly, `set param` shows the value of `param`. The generic command `set param=value` sets up a key and value pair associating the value with the key `param`. You can use parameters to remember discovered values for later use.

- **set dump**

Short form: `s du`

Availability: `always`

Opens the specified dump. The syntax is:

```
set dump[=]<dumpname>
```

After the `set dump` command has run successfully, several additional commands (such as `dis mem` and `dis mmap`) become available. When `set dump` has successfully run (for instance, it was a valid file and it was a dump), another use of `set dump` does nothing. To analyze another dump, start a new `jdmpview` session.

You can also use the `-d` option when starting `jdmpview` to set the dump file.

- **set metadata**

Short form: `s meta`

Availability: `after set dump has run`

Starts the reading of the xml file produced by `jextract`. This activity causes the xml file to be parsed, and gives details about the nature of the dump stored, for use by other commands (such as `dis os` or `dis cls`). The syntax is:

```
set metadata[=]<filename>
```

After `set metadata` has successfully run, subsequent uses do nothing.

- **set workdir**

Short form: `s workdir`

Availability: `always`

Identifies a location to which `jdmpview` can write data. Some commands (such as `dis os` or `trace extract`) create files as part of their function. Typically, these files are created in the same location as the dumpfile; however, sometimes it might be convenient to keep the dumpfile (and the xml) in a read-only location. Its syntax is:

```
set workdir[=]<location>
```

You can also use the `-w` option when starting `jdmpview` to set the working directory.

- **set output**

Short form: `s out`

Availability: `always`

Redirects the output from `jdmpview` to a file rather than to the console (`System.out`). Use it when large amounts of output are expected to be produced from a command (for example, `dis mem 10000,100000`). Its syntax is:

```
set output[=]<location>
```

where `<location>` is either `*` (`System.out`) or `file:filename` (for example, `file:c:\myfile.out`).

You can also use the **-o** option when starting `jdumpview` to set the output location.

- **add output**

Short form: add out

Availability: always

Directs the output from a command to more than one location. Its syntax is:

`add output[=]<location>`

where `<location>` is either `*` (`System.out`) or `file:filename` (for example, `file:c:\myfile.out`).

Working with locks

Use the following commands to work with locks.

- **deadlock**

Availability: after set dump has run

Analyzes the monitors in order to determine if two or more threads are deadlocked.

- **display ls**

Short form: dis ls

Availability: after set dump has run

Displays a summary of the monitors within the JVM.

Showing details of a dump

The following commands show details about the dump.

- **display thread**

Short form: dis t

Availability: after set metadata has run

Gives information about threads in the dumped process. `dis t *` gives information about all the known threads. `dis t` (with no parameters) gives information only about the current thread. `dis t <address of thread>` gives information about a specific thread, the address of which you can obtain from `disproc` or `dis t *`.

- **display sys**

Short form: d sys

Availability: after set metadata has run

Gives information about the dump and the JVM.

- **display ns**

Short form: dis ns

Availability: after set dump has run

This command displays information about the native stack associated with a thread. Information on the Java stack associated with a thread is displayed using `dis t`.

Analyzing the memory

Several commands can be used to display and investigate memory content.

The major content of any dump is the image of memory associated with the process that was dumped. Use the following commands to display and investigate the memory.

Note: These commands do not function until after the `set dump` command has been issued and run successfully.

dis mem

Short form of the `display mem` command.

dis mmap

Short form of the `display mmap` command.

display mem

Displays memory within the dump. The syntax is:

```
display mem <address>[,<numbytes>]
```

<address> is the hex address to display. Optionally, the address is preceded by `0x`. *<numbytes>* is the number of bytes to display. The default is 256 bytes.

display mmap

When a dump is opened using `set dump`, the `jdumpview` command establishes a mapping from the virtual storage ranges held in the dump to their corresponding location in the dump file. `jdumpview` establishes the mapping by creating an internal map. The map is then used by the other memory analysis commands to access information within the memory dump. The `display mmap` command displays the mapping. The display shows you what valid memory ranges are contained within the dump, and their offsets within the dump file.

On z/OS, memory ranges are also associated with an Address Space ID (ASID). In addition to showing memory ranges and their offsets, `display mmap` also shows the ASID to which the memory range belongs. Areas of memory that might seem to be contiguous according to the memory map, or even overlap, are probably not contiguous and therefore have different ASIDs.

find Looks for strings and hex values within the memory dump. The syntax is:
`find pattern[,<start>][,<end>][,<boundary>][,<count>][,<limit>]`

The *<start>* parameter controls where to start the search, *<end>* controls where to end the search, *<boundary>* specifies what byte boundary to use, *<count>* determines how many bytes of memory are displayed when a match is found, and *<limit>* sets a limit of the number of occurrences to report.

fn Short form of the `find` command.

w Short form of the `whatis` command.

whatis

Provides information about what is stored at a memory address. For example, the command can tell you that an address is within an object, in a heap, or within the byte codes associated with a class method. The syntax is:

```
whatis [0x]<hhhhhhhh>
```

<hhhhhhhh> is a hexadecimal memory address to inspect.

Working with classes

Use the following commands to work with classes.

- **display cls** and **display cls <classname>**

*Short form: **dis cls***

*Availability: after **set dump** and **set metadata** have run*

When *classname* is not specified, the command produces a list of all the known classes. The classes are shown with their instance size, and a count of the instances associated with that class (if *dis os* has run). For array classes, the instance size is always 0.

When *<classname>* is specified, and if *dis os* has run, the addresses of all the object instances of that particular class are shown. For classes such as *[char*, where the *[]* indicates that an array class, the number of instances can run into many thousands.

- **display class <classname>**

*Short form: **dis cl <classname>***

*Availability: after **set dump** and **set metadata** have run*

Displays information about the composition of the specified class. The command shows the methods, fields, and static fields associated with the class and other information. This command must not be confused with *dis cls <classname>*.

- **display jitm**

*Short form: **dis jitm***

*Availability: after **set dump** has run*

Displays details about methods that have been processed by the internal JIT.

Working with objects

The following commands allow you to observe and analyze the objects that existed when the dump was taken.

- **display os**

*Short form: **dis os***

*Availability: after **set dump** and **set metadata** have run*

Scans the known heap segments (as identified in the incoming xml metadata) and creates (if necessary) a "jfod" file with information about the object instances found during the scan. It also creates some internal bitmaps that are linked to each heap segment and that indicate the address points in each heap segment that are the starting points of objects.

The output from *dis os* is an object summary that identifies all the classes and gives a count of the number of object instances found and the byte count associated with those instances.

You can run *dis os* only once.

- **display obj address** and **display obj classname**

*Short form: **dis obj***

*Availability: after **set dump**, **set metadata**, and **dis os** have run*

When you specify an *<address>*, displays details about the object at that address. When you specify a *<classname>*, it displays details about all objects of that class. Use the second form with caution because, if there are many instances of the specified class, the output can be large (although you can direct it to an output file for analysis using a file editor).

Generating Heapdumps

Use the following commands to work with Heapdumps.

- **set heapdump** and **set heapdump filename**

*Short form: **s heapdump***

*Availability: after successful **dis os***

Without a parameter, displays the name of the file that was created by the **hd f** command. When you specify the **filename** parameter (for example, **set heapdump c:\my.hd**), the name of the file created by **hd f** is set to the filename you specified. If filename ends in ".gz", the output is produced in gzip compressed format.

The default value for the heapdump filename is *dumpfilename.phd.gz*. For example, if the dump file name (as input to the **set dump** command) is **xyz.20041234.dmp**, the default Heapdump output filename is **xyz.20041234.dmp.phd.gz**.

- **set heapdumpformat**

*Short form: **s heapdumpformat***

*Availability: after successful **dis os***

Sets the format of the output produced. The two settings are **classic** and **portable**. The **classic** option results in a readable text output file. The **portable** option (the default) produces output in a compressed binary format, known as **phd**.

- **set hd_host** and **set hd_port**

*Short form: **s hd_host** and **s hd_port***

*Availability: after successful **dis os***

These two commands control the network host and port that are used for the **hd n** command. The default settings for host and port are **localhost** and **21179** respectively.

- **hd f**

*Availability: after successful **dis os***

Generates heapdump output to a file. Use the **set heapdump** and **set heapdumpformat** commands to control the location and format of the data produced.

- **hd n**

*Availability: after successful **dis os***

Generates heapdump output to a network host. Ensure that you have a receiver running on the host and port specified in the **HD_HOST** and **HD_PORT** options respectively. Also ensure that you have correctly set up any firewall software to allow the connection between your workstation and the host to succeed.

Working with trace

Use the following commands to work with trace.

- **trace extract**

*Availability: after successful **dis os** and **set metadata***

Uses the information in the metadata to extract the trace buffers from the dump and write them to a file (called **extracted.trc**). If no buffers are present in the dump, it displays an error message. The extracted buffers are available for formatting by using the **trace format** command.

- **trace format**

*Availability: after successful **dis os** and **set metadata***

Formats the extracted trace buffers so that they can be viewed using the **trace display** commands. If a **trace extract** has not been issued previously, it is automatically issued by **trace format**.

- **trace display**

*Availability: after successful **dis os** and **set metadata***

Displays the trace output from the trace format command. It displays one page at a time (you can control the page size using the `page display=<size>` command) and allows scrolling through the file using the `trace display +` and `trace display -` commands.

Example session

This example session illustrates a selection of the commands available and their use. The session shown is from a Windows dump. The output from other types of dump is substantially the same. In the example session, some lines have been removed for clarity (and terseness). Some comments (contained within braces) are included to explain various aspects with some comments on individual lines, looking like:

<< comment

User input is in *bold italic*.

{First, invoke `jdumpview` with the name of a dump }

jdumpview sample.dmp

Command Console: " J9 Dump Analysis " << title

Please wait while I process inbound arguments

SET DUMP sample.dmp << command launched on basis of inbound argument
Recognised as a 32-bit little-endian windows dump. << dump exists
and is supported

Trying to use "sample.dmp.xml" as metadata.....
Issuing "SET METADATA sample.dmp.xml" << work with the xml
Parsing of xml started for file dl2.dmp.xml... be patient
Warning: "systemProperties" is an unknown tag and is being ignored
Warning: "property" is an unknown tag and is being ignored

Parsing ended

Ready....('h' shows help, 'cmds' shows available commands) << `jdumpview`
is ready to accept user input

{ the output produced by `h` (or `help`) is illustrated below – "`help`
`<command_name>`" gives information on a specific command

h
General Help
=====

To see what commands are available use the "`cmds`" command.
Note: The available command set can change as a result of some actions
- such as "`set dump`" or "`set metadata`".

The general form of a command is `NOUN VERB PARM1 [,PARM2] ... [PARMn]`
Note: some commands do not need a verb or parameters. The command parser
strips "=" characters and brackets from the input - this allows
alternative command formats like "`set dump=c:\mydump.dmp`" to work.

Use "`help command`" to obtain more help on a specific command

Ready....

help set dump

This command is usually one of the first commands entered. It requires a file

name as a parameter. The file identified (presuming it exists) is verified to be a dump, its type established, and the dump analysed to establish a memory map (see "dis mmap" for more details).

Note: as an alternative to using set dump then starting jdmpview with a parameter of "-ddumpname" (note no space between the -d and filename) or with just the filename will open the dump before the first "Ready...." appears.

As part of the processing when "set dump" is issued then if an xml file (as produced out of jextract) is found matching the dump then a "set metadata" command will be issued.

Ready....

{ The next command "dis os" is covered below. This command scans the heap segments that were identified in the xml and produces a names index file (.jfod) to allow subsequent analysis of objects. For large dumps with several millions of objects then this command could take a long time. }

dis os

Names index file in use is: sample.dmp.jfod

Heap Summary
=====

WARNING: It can take a long time to traverse the heaps!!!! - Please be patient
Recording class instances

... 686 class instances recorded

Starting scan of heap segment 0 start=0x420000 end=0x4207e8
object count= 47
Starting scan of heap segment 1 start=0x420800 end=0x421898
object count= 85

==== lines removed for terseness =====

==== lines removed for terseness =====

Starting scan of heap segment 3655 start=0x17cb000 end=0x17eafb0
object count= 2513
Starting scan of heap segment 3656 start=0x17eb000 end=0x180afe0
object count= 2517
Starting scan of heap segment 3657 start=0x180b000 end=0x180dd80
object count= 223

Object Summary

[[java/lang/String has 1 instances (total size= 32)
[[java/lang/ref/SoftReference has 1 instances (total size= 24)
[boolean has 9 instances (total size= 1683)
[byte has 989 instances (total size= 2397336)
[char has 153683 instances (total size= 7991516)
[com/ibm/jvm/j9/dump/command/Command has 5 instances (total size= 245)
==== lines removed for terseness =====
==== lines removed for terseness =====
sun/reflect/MethodAccessorGenerator\$1 has 1 instances (total size= 28)
sun/reflect/NativeConstructorAccessorImpl has 1 instances (total size= 24)
sun/reflect/NativeMethodAccessorImpl has 24 instances (total size= 576)
sun/reflect/ReflectionFactory has 1 instances (total size= 12)
sun/reflect/ReflectionFactory\$1 has 1 instances (total size= 12)
sun/reflect/ReflectionFactory\$GetReflectionFactoryAction has 1 instances
(total size=12)
sun/security/action/GetPropertyAction has 32 instances (total size= 640)

```
sun/security/action/LoadLibraryAction has 3 instances (total size= 48)
sun/security/util/ManifestEntryVerifier$1 has 1 instances (total size= 12)
```

```
Total number of objects = 485261
Total size of objects    = 20278079 bytes
```

```
Total locked objects    = 0
```

```
Ready ....
```

```
{ The next command illustrated is "dis sys". This shows some generic information
about the dump being processed }
```

dis sys

```
System Summary
=====
```

```
32bit
Little Endian (i.e. 0x12345678 in memory could be 0x78563412 if it was a pointer)
System Memory = 2146353152 Bytes
System       : windows
Java Version  : 2.3 Windows XP
BuildLevel    : 20050926_03412_1HdSMR
Uuid         : 16742003772852651681
```

```
Ready ....
```

```
{ The next command illustrated is "cmds" – this shows the syntax of the
currently recognised commands }
```

cmds

```
Known Commands
=====
```

```
SET DUMP (Identifies the dump to work with)
SET METADATA (Identifies xml metadata file - rarely needed)
QUIT (Terminates jdmpview session)
HELP * (Provides generic and specific help)
==== lines removed for terseness =====
DIS MMAP (Displays the virtual address ranges within the dump)
DIS MEM (Formats and displays portions of memory)
==== lines removed for terseness =====
```

```
"help command" shows details of each command
```

```
Note: some supported commands may not be shown in the above
list as they only become available after successful issuance
of other commands (such as "set dump" or "dis os")
```

```
Ready....
```

```
{ dis proc can be used to show process information and includes details about
the commandline that was issued, the threads (both java and native) seen, the
environment variable settings in the process and the positions in memory of
loaded modules}
```

dis proc

```

Process Information
=====
Architecture: 32bit - Little Endian
Thread: 0x3b1300 Thread name: main
Thread: 0x3b1700 Thread name: ** Not a Java Thread **
Thread: 0x3b1b00 Thread name: Signal Dispatcher
Thread: 0x41fcdb00 Thread name: Finalizer thread
Thread: 0xc70 Thread name: Un-established
Thread: 0xb54 Thread name: Un-established
Thread: 0xb34 Thread name: Un-established
Thread: 0xfe0 Thread name: Un-established
Thread: 0xae0 Thread name: Un-established
Thread: 0xf98 Thread name: Un-established
Thread: 0xe38 Thread name: Un-established
Thread: 0xe2c Thread name: Un-established

CommandLine = jdmpview -J-Xdump:system:events=fullgc << the command line

Environment Variables
=====

tvlogsessioncount=5000
PATH=C:\sdk2709\jre\bin;C:\Perl\bin\;C:\Program Files\IBM\Infoprint Select;...
PD_SOCKET=6874
==== lines removed for terseness =====
ALLUSERSPROFILE=C:\Documents and Settings\All Users
VS71COMNTOOLS=C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Tools\

Loaded Information
=====

C:\sdk2709\bin\jdmpview.exe
  at 0x400000 length=81920(0x14000)
C:\WINDOWS\System32\ntdll.dll
  at 0x77f50000 length=684032(0xa7000)
C:\WINDOWS\system32\kernel32.dll
  at 0x77e60000 length=942080(0xe6000)
  ==== lines removed for terseness =====
  ==== lines removed for terseness =====
C:\sdk2709\jre\bin\java.dll
  at 0x417f0000 length=180224(0x2c000)
C:\sdk2709\jre\bin\wrappers.dll
  at 0x3e0000 length=32768(0x8000)
C:\WINDOWS\System32\Secur32.dll
  at 0x76f90000 length=65536(0x10000)
  ==== lines removed for terseness =====
  ==== lines removed for terseness =====
C:\WINDOWS\system32\VERSION.dll
  at 0x77c00000 length=28672(0x7000)
C:\WINDOWS\System32\PSAPI.DLL
  at 0x76bf0000 length=45056(0xb000)

Ready ....

{ dis mmap is used to display the available memory ranges }

help dis mmap

This command shows the memory ranges available from the dump, together with the
offset of that memory within the dump file. In the case of zOS it also shows the
address space associated with each range.

Ready ....

dis mmap

```

Memory Map

=====

```
Addr: 0x00010000 Size: 0x2000 (8192) File Offset: 0x4b64 (19300)
Addr: 0x00020000 Size: 0x1000 (4096) File Offset: 0x6b64 (27492)
Addr: 0x0006c000 Size: 0x5000 (20480) File Offset: 0x7b64 (31588)
Addr: 0x00080000 Size: 0x106000 (1073152) File Offset: 0xcb64 (52068)
==== lines removed for terseness =====
==== lines removed for terseness =====
Addr: 0x7ffb0000 Size: 0x24000 (147456) File Offset: 0x3001b64 (50338660)
Addr: 0x7ffd7000 Size: 0xa000 (40960) File Offset: 0x3025b64 (50486116)
```

Ready

{ dis mem can be used to look directly at memory in the dump, in the example below we are looking at the start of the memory associated with java.exe as found from the loaded module information displayed previously }

dis mem 400000

```
00400000: 4D5A9000 03000000 04000000 FFFF0000 | MZ.....
00400010: B8000000 00000000 40000000 00000000 | .....@.....
00400020: 00000000 00000000 00000000 00000000 | .....
00400030: 00000000 00000000 00000000 F8000000 | .....
00400040: 0E1FBA0E 00B409CD 21B8014C CD215468 | .....!.L!Th
00400050: 69732070 726F6772 616D2063 616E6E6F | is program canno
00400060: 74206265 2072756E 20696E20 444F5320 | t be run in DOS
00400070: 6D6F6465 2E0D0D0A 24000000 00000000 | mode...$.
00400080: 706C875B 340DE908 340DE908 340DE908 | pl.[4...4...
00400090: 27058008 3F0DE908 3101B408 360DE908 | '...?...1...6...
004000a0: 31018908 350DE908 3101E608 300DE908 | 1...5...1...0...
004000b0: B705B408 310DE908 340DE808 710DE908 | ...1...4...q...
004000c0: 3101B608 2F0DE908 D806B708 350DE908 | 1.../.....5...
004000d0: 3101B308 350DE908 52696368 340DE908 | 1...5...Rich4...
004000e0: 00000000 00000000 00000000 00000000 | .....
004000f0: 00000000 00000000 50450000 4C010500 | .....PE..L...
```

Ready

{ the dis cls command when used without an additional parameter displays information on all the loaded classes whilst with the addition of a classname it shows the addresses of all instances of that class in the dump ... }

dis cls [boolean

[boolean instance size=0 object count=9

```
0x88f898 0x884088 0x87e128 0x4cca58 0x4cc848 0x4cc6f0
0x4c7c50 0x4c7640 0x47b1d0
```

Ready

{ ... and dis mem can be used to look directly at the memory ... }

dis mem 0x88f898

```
0088f898: C03A4541 0380263E 00000000 80000000 | .:EA..&>.....
0088f8a8: 01010101 01010101 01010101 01010101 | .....
0088f8b8: 01010101 01010101 01010101 01010101 | .....
0088f8c8: 01000101 00010000 00000000 00000000 | .....
0088f8d8: 00000000 00000000 00000000 01000100 | .....
0088f8e8: 00000000 00000000 00000000 00000000 | .....
0088f8f8: 00000000 00000000 00000001 01010100 | .....
0088f908: 01000000 00000000 00000000 00000000 | .....
```

0088f918:	00000000	00000000	00000001	01010101
0088f928:	083C4541	05804A3E	00000000	80000000	.<EA..J>.....
0088f938:	30003000	30003000	30003000	30003000	0.0.0.0.0.0.0.
0088f948:	30003000	30003000	30003000	30003000	0.0.0.0.0.0.0.
0088f958:	31003100	31003100	31003100	31003100	1.1.1.1.1.1.1.
0088f968:	31003100	31003100	31003100	31003100	1.1.1.1.1.1.1.
0088f978:	32000000	32003200	00003200	00000000	2...2.2...2....
0088f988:	00000000	00000000	00000000	00000000

Ready

{ or dis obj to give a formatted display of the object based on a combination of the class information and the instance data }

dis obj 0x88f898

```
[boolean@0x88f898
Its an Array object: primitive: filled with - [boolean
    instance size = 144
    128 elements , element size = 1
    arity is 1
```

Ready

dis cls sun/io/ByteToCharUTF8

```
sun/io/ByteToCharUTF8    instance size=52    object count=1

    0x4b1950
```

Ready

dis obj 0x4b1950

```
sun/io/ByteToCharUTF8@0x4b1950

    (16) fieldName: subMode  sig: Z  value= TRUE (0x1)
    (12) fieldName: subChars sig: [C Its a primitive array @0x4b1988
    (20) fieldName: charOff  sig: I  value=0 (0x0)
    (24) fieldName: byteOff  sig: I  value=0 (0x0)
    (28) fieldName: badInputLength sig: I  value=0 (0x0)
    (36) fieldName: state    sig: I  value=0 (0x0)
    (40) fieldName: inputSize sig: I  value=0 (0x0)
    (44) fieldName: value    sig: I  value=0 (0x0)
    (32) fieldName: bidiParms sig: Ljava/lang/String;
           value= 0x435ba8 ==> "NO"
    (48) fieldName: bidiEnabled sig: Z  value= FALSE (0x0)
```

Ready

{ the dis cl command is used to show information about a particular class, its fields, statics and methods }

dis cl sun/io/ByteToCharUTF8

```
name = sun/io/ByteToCharUTF8 id = 0x41df0dc8 superId = 0x4147fc60
instanceSize = 52 loader = 0x17a98c
modifiers: public super
Inheritance chain....
    java/lang/Object
        sun/io/ByteToCharConverter
            sun/io/ByteToCharUTF8

Fields.....
subMode modifiers: protected sig: Z offset: 16 (defined in class
                                                0x4147fc60)
```

```

subChars  modifiers: protected  sig: [C  offset: 12  (defined in class
                                                0x4147fc60)
charOff   modifiers: protected  sig: I  offset: 20  (defined in class
                                                0x4147fc60)
byteOff   modifiers: protected  sig: I  offset: 24  (defined in class
                                                0x4147fc60)
badInputLength modifiers: protected sig: I offset: 28 (defined in class
                                                0x4147fc60)

state modifiers: private sig: I  offset: 36
inputSize modifiers: private sig: I  offset: 40
value modifiers: private sig: I  offset: 44
bidiParms modifiers: private sig:Ljava/lang/String; offset: 32
bidiEnabled modifiers: private sig: Z  offset: 48

Statics.....
    name: States modifiers: public static final value: 0x4b1860
                                                sig: [I
    name: StateMask modifiers: public static final value: 0x4b18f0
                                                sig: [I
    name: bidiInit modifiers: private static value: 0x435ba8 sig:
                                                Ljava/lang/String;

Methods.....
    name: <init> sig: ()V id: 0x41df0ec0 modifiers: public
        Bytecode start=0x41e0f884 end=0x41e0f910
    name: flush sig: ([CII)I id: 0x41df0ed0 modifiers: public
        Bytecode start=0x41e0f924 end=0x41e0f948
    name: setException sig: (II)V id: 0x41df0ee0 modifiers: private
        Bytecode start=0x41e0f964 end=0x41e0f9a4
    name: convert sig: ([BII[CII)I id: 0x41df0ef0 modifiers: public
        Bytecode start=0x41e0f9b8 end=0x41e0fc60
    name: doBidi sig: ([CII)V id: 0x41df0f00 modifiers:
        Bytecode start=0x41e0fc80 end=0x41e0fccc
    name: getCharacterEncoding sig: ()Ljava/lang/String;
        id: 0x41df0f10 modifiers: public
        Bytecode start=0x41e0fce0 end=0x41e0fce4
    name: reset sig: ()V id: 0x41df0f20 modifiers: public
        Bytecode start=0x41e0fcf8 end=0x41e0fd08
    name: <clinit> sig: ()V id: 0x41df0f30 modifiers: static
        Bytecode start=0x41e0fd1c end=0x41e0fdcc

```

Ready

{ dis t shows the available information for the current thread
Note that the "set thread" command can be used to change the current thread and
"dis t *" can be used to see all the threads at once }

dis t

Info for thread - 0x3b1300

=====

```

Name      : main
Id        : 0x3b1300
NativeId: 0xc70
Obj       : 0x420500 (java/lang/Thread)
State    : Running
Stack:
    method: java/lang/Long::toUnsignedString(JI)Ljava/lang/String;
                                                pc: 0x415bd735
    arguments: 0x41e32dec
    method: java/lang/Long::toHexString(J)Ljava/lang/String;
                                                pc: 0x415be110
    arguments: 0x41e32df8

```

==== lines removed for terseness =====
==== lines removed for terseness =====

```
method: com/ibm/jvm/j9/dump/commandconsole/J9JVMConsole:: .....
method: com/ibm/jvm/j9/dump/commandconsole/J9JVMConsole:<init>..
method: com/ibm/jvm/j9/dump/commandconsole/J9JVMConsole::main.....
```

Ready

{ the next section of commands shows some features of the find commands }

find java

Note: your search result limit was 1 ... there may be more results

00083bfb:	6A617661	5C736F76	5C6A6176	612E7064	java\sov\java.pd
00083c0b:	62000000	0043000B	00000109	00000000	b....C.....
00083c1b:	00010000	01010000	01000100	00010100
00083c2b:	00010000	01010101	01000001	00010001
00083c3b:	00000101	00010100	00010101	00000100
00083c4b:	00010000	01000100	00010100	00010000
00083c5b:	03000003	00000100	01010001	01000001
00083c6b:	01000003	00000100	00010001	00000101
00083c7b:	00000100	00030000	03000001	00010100
00083c8b:	01010000	01030100	00010103	00000101
00083c9b:	00000100	01000001	01000001	00010000
00083cab:	01000100	00010000	01000003	00000101
00083cbb:	00000101	01000100	00010001	00000100
00083ccb:	03000001	00010000	01010000	01010100
00083cdb:	00010001	00000100	03010003	00010000
00083ceb:	01000300	00010000	01000101	00010000

Tip 1: Use FINDNEXT (FN) command to progress through them

Tip 2: Use "SET FINDMODE=V" to do automatic WHATIS

Find finished...

Ready

help find

Help for Find

=====

The find command allows memory in the dump to be searched for strings and hex patterns. The syntax is:

Find pattern[,start][,end][,boundary][,count][,limit]

Examples are:

Find java -

 this would search all available memory for the string "java"

Note: only matches with the right case will be found. Only 1 match will be displayed (default for limit) and the first 256 bytes (default for count) of the first match will be displayed.

Find 0xF0F0F0,804c000,10000000,8,32,50

 this would search for the hex string "F0F0F0" in memory range 0x804c000 thru 0x10000000.

 Only matches starting on an 8 byte boundary would count and the first 32 bytes of the first match will be displayed. Up to 50 matches will be displayed.

** The FINDNEXT command (FN) will repeat the previous FIND command but starting just beyond the result of the previous search.

There is also a FINDPTR (FP) command that will accept a normalised address and issue the appropriate FIND command having adjusted the address to the endianness bitness of the dump in use.

** If you want to search EBCDIC dumps(zOS) for ascii text then you can use the

"SET FORMATAS=A" command to force the search to assume that the search text is ASCII (and "SET FORMATAS=E" would allow the opposite).

Ready

fn

issuing FIND java,83bfc,ffffffffffffffff,1,256,1

Note: your search result limit was 1 ... there may be more results

00083c04:	6A617661	2E706462	00000000	43000B00	java.pdb....C...
00083c14:	00010900	00000000	01000001	01000001
00083c24:	00010000	01010000	01000001	01010101
00083c34:	00000100	01000100	00010100	01010000
00083c44:	01010100	00010000	01000001	00010000
00083c54:	01010000	01000003	00000300	00010001
00083c64:	01000101	00000101	00000300	00010000
00083c74:	01000100	00010100	00010000	03000003
00083c84:	00000100	01010001	01000001	03010000
00083c94:	01010300	00010100	00010001	00000101
00083ca4:	00000100	01000001	00010000	01000001
00083cb4:	00000300	00010100	00010101	00010000
00083cc4:	01000100	00010003	00000100	01000001
00083cd4:	01000001	01010000	01000100	00010003
00083ce4:	01000300	01000001	00030000	01000001
00083cf4:	00010100	01000001	00010300	01000001

Tip 1: Use FINDNEXT (FN) command to progress through them

Tip 2: Use "SET FINDMODE=V" to do automatic WHATIS

Find finished...

Ready

fp

No parameter specified -

Help for FindPtr

=====

The findptr command (findp and fp are used as short forms) takes a hex value which is assumed to be a pointer (8 bytes on 32 bit platforms, 16 bytes on 64 bit platforms) and searches memory looking for it. The syntax is based on that for find:

Findp pointer[,start][,end][,boundary][,count][,limit]

Examples are:

Findp 0x10000 -

this would search all available memory for the pointer 0x10000

Note: jdmpview adjusts the input command to match the endianness and pointer size of the dump. Thus for a windows 32-bit system the above example would be mapped to "find 0x00000100,,4" to take account of little endianness and 32-bitness (i.e. pointers are assumed to be aligned on 32 bit boundaries).

Ready

fp 6176616a

Note: your search result limit was 1 ... there may be more results

00083bfb:	6A617661	5C736F76	5C6A6176	612E7064	java\sov\java.pd
00083c0b:	62000000	0043000B	00000109	00000000	b....C.....

00083c1b:	00010000	01010000	01000100	00010100
00083c2b:	00010000	01010101	01000001	00010001
00083c3b:	00000101	00010100	00010101	00000100
00083c4b:	00010000	01000100	00010100	00010000
00083c5b:	03000003	00000100	01010001	01000001
00083c6b:	01000003	00000100	00010001	00000101
00083c7b:	00000100	00030000	03000001	00010100
00083c8b:	01010000	01030100	00010103	00000101
00083c9b:	00000100	01000001	01000001	00010000
00083cab:	01000100	00010000	01000003	00000101
00083cbb:	00000101	01000100	00010001	00000100
00083ccb:	03000001	00010000	01010000	01010100
00083cdb:	00010001	00000100	03010003	00010000
00083ceb:	01000300	00010000	01000101	00010000

Tip 1: Use FINDNEXT (FN) command to progress through them

Tip 2: Use "SET FINDMODE=V" to do automatic WHATIS

Find finished...

Ready

Chapter 25. Tracing Java applications and the JVM

JVM trace is a trace facility that is provided in all IBM-supplied JVMs with minimal affect on performance. In most cases, the trace data is kept in a compact binary format, that can be formatted with the Java formatter that is supplied.

Tracing is enabled by default, together with a small set of trace points going to memory buffers. You can enable tracepoints at runtime by using levels, components, group names, or individual tracepoint identifiers.

This chapter describes JVM trace in:

- “What can be traced?”
- “Default tracing” on page 284
- “Where does the data go?” on page 285
- “Controlling the trace” on page 287
- “Determining the tracepoint ID of a tracepoint” on page 305
- “Application trace” on page 306
- “Using method trace” on page 309

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.yyymmdd.hhmmssstt.process.trc`.

You must also think carefully about which components need to be traced and what level of tracing is required. For example, if you are tracing a suspected shared classes problem, it might be enough to trace all components at level 1, and j9shr at level 9, while maximal can be used to show parameters and other information for the failing component.

Types of tracepoint

There are two types of tracepoints inside the JVM: regular and auxiliary.

Regular tracepoints

Regular tracepoints include:

- method tracepoints
- application tracepoints
- data tracepoints inside the JVM
- data tracepoints inside class libraries

You can display regular tracepoint data on the screen or save the data to a file. You can also use command line options to trigger specific actions when regular tracepoints fire. See the section “Detailed descriptions of trace options” on page 288 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}
```

CAUTION:

If you specify any -Xtrace options on the command line, the default trace options are disabled.

The data generated by those tracepoints is continuously captured in wrapping, per thread memory buffers. (For information about specific options, see “Detailed descriptions of trace options” on page 288.)

You can find tracepoint information in the following diagnostics data:

- System memory dumps, extracted using `jdmpview`.
- Snap traces, generated when the JVM encounters a problem or an output file is specified. Chapter 21, “Using dump agents,” on page 223 describes more ways to create a snap trace.

- For exception trace only, in Javadumps.

Default memory management tracing

The default trace options are designed to ensure that Javadumps always contain a record of the most recent memory management history, regardless of how much work the JVM has performed since the garbage collection cycle was last called.

The **exception=j9mm{gclogger}** clause of the default trace set specifies that a history of garbage collection cycles that have occurred in the JVM is continuously recorded. The **gclogger** group of tracepoints in the j9mm component constitutes a set of tracepoints that record a snapshot of each garbage collection cycle. These tracepoints are recorded in their own separate buffer, called the exception buffer. The effect is that the tracepoints are not overwritten by the higher frequency tracepoints of the JVM.

The **GC History** section of the Javadump is based on the information in the exception buffer. If a garbage collection cycle has occurred in a traced JVM, the Javadump probably contains a **GC History** section.

Default assertion tracing

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

The JVM continues running after the logs have been produced. Assertion failures often indicate a serious problem and the JVM might exit with a subsequent error. Even if the JVM does not encounter another error, restart the JVM as soon as possible. Send a service request to IBM, including the standard error output and the .trc and .dmp files produced.

When an assertion trace point is 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 using the standard trace command-line options. See “Controlling the trace” on page 287 for more details.

Where does the data go?

Trace data can be written to a number of locations.

Trace data can go into:

- Memory buffers that can be dumped or snapped when a problem occurs
- One or more files that are using buffered I/O
- An external agent in real time
- stderr in real time

- Any combination of the above

Writing trace data to memory buffers

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

Buffers are allocated on a per-thread principle. This principle removes contention between threads, and prevents trace data for an individual thread from being mixed in with trace data from other threads. For example, if one particular thread is not being dispatched, its trace information is still available when the buffers are dumped or snapped. Use the **-Xtrace:buffers=<size>** option to control the size of the buffer allocated to each thread.

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

To examine the trace data captured in these memory buffers, you must snap or dump the data, then format the buffers.

Snapping buffers

Under default conditions, a running JVM collects a small amount of trace data in special wraparound buffers. This data is sent to a snap trace file under certain conditions:

- An uncaught `OutOfMemoryError` occurs.
- An operating system signal or exception occurs.
- The `com.ibm.jvm.Trace.snap()` Java API is called.
- The JVMRI `TraceSnap` function is called.

The resulting snap trace file is placed into the current working directory, with a name in the format `Snapnnnn.yyyymmdd.hhmmss.th.process.trc`, where `nnnn` is a sequence number reset to 0001 at JVM startup, `yyymmdd` is the current date, `hhmmss.th` is the current time, and `process` is the process identifier. This file is in a binary format, and requires the use of the supplied trace formatter so that you can read it.

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

Extracting buffers from system dump

You can extract the buffers from a system dump core file by using the Dump Viewer.

Writing trace data to a file

You can write trace data to a file continuously as an extension to the in-storage trace, but, instead of one buffer per thread, at least two buffers per thread are allocated, and the data is written to the file before wrapping can occur.

This allocation allows the thread to continue to run while a full trace buffer is written to disk. Depending on trace volume, buffer size, and the bandwidth of the output device, multiple buffers might be allocated to a given thread to keep pace with trace data that is being generated.

A thread is never stopped to allow trace buffers to be written. If the rate of trace data generation greatly exceeds the speed of the output device, excessive memory

usage might occur and cause out-of-memory conditions. To prevent this, use the **nodynamic** option of the **buffers** trace option. For long-running trace runs, a **wrap** option is available to limit the file to a given size. It is also possible to create a sequence of files when the trace output will move back to the first file once the sequence of files are full. See the **output** option for details. You must use the trace formatter to format trace data from the file.

Because trace data is buffered, if the JVM does not exit normally, residual trace buffers might not be flushed to the file. If the JVM encounters a fatal error, the buffers can be extracted from a system dump if that is available. When a snap file is created, all available buffers are always written to it.

External tracing

You can route trace to an agent by using JVMRI TraceRegister.

This mechanism allows a callback routine to be called immediately when any of the selected tracepoints is found without buffering the trace results. The trace data is in raw binary form. Further details can be found in the JVMRI section.

Tracing to stderr

For lower volume or non-performance-critical tracing, the trace data can be formatted and routed to stderr immediately without buffering.

For more information, see “Using method trace” on page 309.

Trace combinations

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

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

Controlling the trace

You have several ways by which you can control the trace.

You can control the trace in several ways by using:

- The **-Xtrace** options when launching the JVM, including trace trigger events
- A trace properties file
- `com.ibm.jvm.Trace API`
- JVMTI and JVMRI from an external agent

Note:

1. By default, trace options equivalent to the following are enabled:
`-Xtrace:maximal=all{level1},exception=j9mm{gclogger}`
2. Many diagnostic tools start a JVM. When using the **IBM_JAVA_OPTIONS** environment variable trace to a file, starting a diagnostic tool might overwrite the trace data generated from your application. Use the command-line tracing

options or add %d, %p or %t to the trace file name to prevent this from happening. See “Detailed descriptions of trace options” 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`

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:

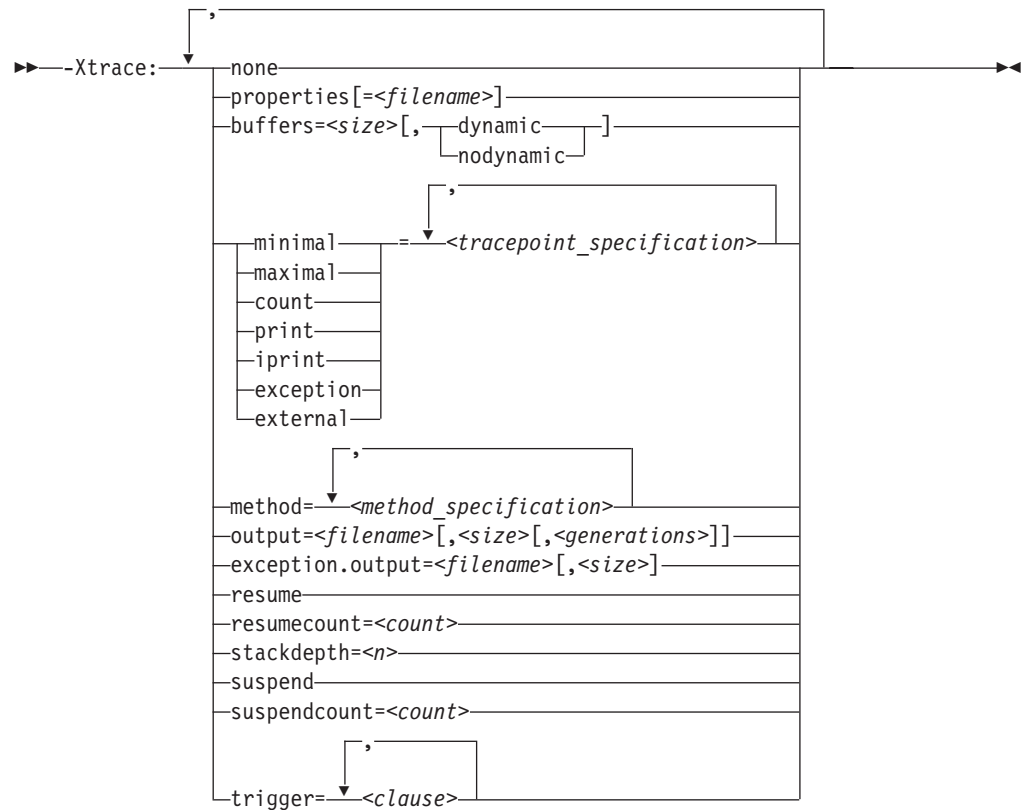
`java "-Xtrace:<name>,<another_name>=<value>" HelloWorld`

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



none

Disables tracing

Example

- Tracing disabled:
-Xtrace:none

properties[=<filename>]

You can use properties files to control trace. A properties file saves typing and, over time, causes a library of these files to be created. Each file is tailored to solving problems in a particular area.

This trace option allows you to specify in a file any of the other trace options, thereby reducing the length of the invocation command-line. The format of the file is a flat ASCII file that contains trace options. If *<filename>* is not specified, a default name of `IBMTRACE.properties` is searched for in the current directory. Nesting is not supported; that is, the file cannot contain a `properties` option. If any error is found when the file is accessed, JVM initialization fails with an explanatory error message and return code. All the options that are in the file are processed in the sequence in which they are stored in the file, before the next option that is obtained through the normal mechanism is processed. Therefore, a command-line property always overrides a property that is in the file.

An existing restriction means that properties that take the form *<name>=<value>* cannot be left to default if they are specified in the property file; that is, you must specify a value, for example `maximal=all`.

Another restriction means that properties files are sensitive to white space. Do not add white space before, after, or within the trace options.

You can make comments as follows:

```
// This is a comment. Note that it starts in column 1
```

Examples

- Use IBMTRACE.properties in the current directory:
-Xtrace:properties
- Use trace.prop in the current directory:
-Xtrace:properties=trace.prop
- Use c:\trc\gc\trace.props:
-Xtrace:properties=c:\trc\gc\trace.props

Here is an example property file:

```
minimal=all  
// maximal=j9mm  
maximal=j9shr  
buffers=20k  
output=c:\traces\classloader.trc  
print=tpnid(j9vm.23-25)
```

buffers=nnnk|nnnm[,dynamic|nodynamic]

You can modify the size of the buffers to change how much diagnostics output is provided in a snap dump. This buffer is allocated for each thread that makes trace entries.

From Java 5 SR 10, you do not need to specify the buffer size.

The trace option can be specified in two ways:

- **buffers=dynamic|nodynamic**
- **buffers=nnnk|nnnm[,dynamic|nodynamic]**

If external trace is enabled, the number of buffers is doubled; that is, each thread allocates two or more buffers. The same buffer size is used for state and exception tracing, but, in this case, buffers are allocated globally. The default is 8 KB per thread.

The **dynamic** and **nodynamic** options have meaning only when tracing to an output file. If **dynamic** is specified, buffers are allocated as needed to match the rate of trace data generation to the output media. Conversely, if **nodynamic** is specified, a maximum of two buffers per thread is allocated. The default is **dynamic**. The dynamic option is effective only when you are tracing to an output file.

Note: If **nodynamic** is specified, you might lose trace data if the volume of trace data that is produced exceeds the bandwidth of the trace output file. Message UTE115 is issued when the first trace entry is lost, and message UTE018 is issued at JVM termination.

Examples

- Dynamic buffering with increased buffer size of 2 MB per thread:
-Xtrace:buffers=2m

or in a properties file:

```
buffers=2m
```

- Trace buffers limited to two buffers per thread, each of 128 KB:

```
-Xtrace:buffers={128k,nodynamic}
```

or in a properties file:

```
buffers=128k,nodynamic
```

- Trace using default buffer size of 8 KB, limited to two buffers per thread (Java 5 SR 10 or later):

```
-Xtrace:buffers=nodynamic
```

or in a properties file:

```
buffers=nodynamic
```

Options that control tracepoint activation

These options control which individual tracepoints are activated at runtime and the implicit destination of the trace data.

In some cases, you must use them with other options. For example, if you specify maximal or minimal tracepoints, the trace data is put into memory buffers. If you are going to send the data to a file, you must use an output option to specify the destination filename.

```
minimal=[!]<tracepoint_specification>[...]
```

```
maximal=[!]<tracepoint_specification>[...]
```

```
count=[!]<tracepoint_specification>[...]
```

```
print=[!]<tracepoint_specification>[...]
```

```
iprint=[!]<tracepoint_specification>[...]
```

```
exception=[!]<tracepoint_specification>[...]
```

```
external=[!]<tracepoint_specification>[...]
```

```
none[=<tracepoint_specification>[...]]
```

Note that all these properties are independent of each other and can be mixed and matched in any way that you choose.

From IBM SDK 5.0 SR10, you must provide at least one tracepoint specification when using the **minimal**, **maximal**, **count**, **print**, **iprint**, **exception** and **external** options. In some older versions of the SDK the tracepoint specification defaults to 'all'.

Multiple statements of each type of trace are allowed and their effect is cumulative. To do this, you must use a trace properties file for multiple trace options of the same name.

minimal and maximal

minimal and **maximal** trace data is placed into internal trace buffers that can then be written to a snap file or written to the files that are specified in an output trace option. The **minimal** option records only the timestamp and tracepoint identifier. When the trace is formatted, missing trace data is replaced with the characters "???" in the output file. The **maximal** option specifies that all associated data is traced. If a tracepoint is activated by both trace options, **maximal** trace data is produced. Note that these types of trace are completely independent from any types that follow them. For example, if the **minimal** option is specified, it does not affect a later option such as **print**.

count

The **count** option requests that only a count of the selected tracepoints is kept. At JVM termination, all non-zero totals of tracepoints (sorted by tracepoint id) are written to a file, called `utTrcCounters`, in the current directory. This information is useful if you want to determine the overhead of particular tracepoints, but do not want to produce a large amount (GB) of trace data.

For example, to count the tracepoints used in the default trace configuration, use the following:

```
-Xtrace:count=all{!level1},count=j9mm{gclogger}
```

print

The **print** option causes the specified tracepoints to be routed to `stderr` in real-time. The JVM tracepoints are formatted using `J9TraceFormat.dat`. The class library tracepoints are formatted by `TraceFormat.dat`. `J9TraceFormat.dat` and `TraceFormat.dat` are shipped in `sdk/jre/lib` and are automatically found by the runtime.

iprint

The **iprint** option is the same as the **print** option, but uses indenting to format the trace.

exception

When **exception** trace is enabled, the trace data is collected in internal buffers that are separate from the normal buffers. These internal buffers can then be written to a snap file or written to the file that is specified in an **exception.output** option.

The **exception** option allows low-volume tracing in buffers and files that are distinct from the higher-volume information that **minimal** and **maximal** tracing have provided. In most cases, this information is exception-type data, but you can use this option to capture any trace data that you want.

This form of tracing is channeled through a single set of buffers, as opposed to the buffer-per-thread approach for normal trace, and buffer contention might occur if high volumes of trace data are collected. A difference exists in the `<tracepoint_specification>` defaults for exception tracing; see "Tracepoint specification" on page 293.

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 284. You can send additional tracepoints to the exception buffers or switch off the garbage collection tracepoints. Changing the exception trace buffers will alter the contents of the **GC History** section in any Javadumps.

Note: When **exception** trace is entered for an active tracepoint, the current thread id is checked against the previous caller's thread id. If it is a different thread, or this is the first call to **exception** trace, a context tracepoint is put into the trace buffer first. This context tracepoint consists only of the current thread id. This is necessary because of the single set of buffers for exception trace. (The formatter identifies all trace entries as coming from the "Exception trace pseudo thread" when it formats **exception** trace files.)

external

The **external** option channels trace data to registered trace listeners in real-time. JVMRI is used to register or deregister as a trace listener. If no listeners are registered, this form of trace does nothing except waste machine cycles on each activated tracepoint.

Examples

- Default options applied:

```
java
```
- No effect apart from ensuring that the trace engine is loaded (which is the default behavior):

```
java -Xtrace
```
- Trace engine is not loaded:

```
java -Xtrace:none
```
- Printing for j9vm.209 only:

```
java -Xtrace:iprint=tpnid{j9vm.209}
```

Tracepoint specification:

You enable tracepoints by specifying *component* and *tracepoint*.

If no qualifier parameters are entered, all tracepoints are enabled, except for **exception.output** trace, where the default is **all {exception}**.

The *<tracepoint_specification>* is as follows:

[!]<component>[<type>] or [!]tpnid<tracepoint_id>[,...]

where:

! is a logical not. That is, the tracepoints that are specified immediately following the ! are turned off.

<component>

is a Java subcomponent, as detailed in Table 1. To include all Java subcomponents, specify **all**.

Table 1. Java subcomponents

Subcomponent name	Description
audio	Class library com.sun.media.sound native code
awt	Class library AWT native code
fontmanager	Class library AWT font manager native code
j9bcu	VM byte code utilities
j9bcverify	VM byte code verification
j9decomp	VM byte code run time
j9dbgtspt	VM debug transport
j9dmp	VM dump
j9jcl	VM class libraries
j9jit	VM JIT interface
j9jvmti	VM JVMTI support
j9mm	VM memory management
j9prt	VM port library
j9scar	VM class library interface
j9shr	VM shared classes
j9trc	VM trace
j9vm	VM general

Table 1. Java subcomponents (continued)

Subcomponent name	Description
j9vrb	VM verbose stack walker
java	Class library general native code (including java.io)
mawt	Class library AWT motif native code
mt	Java methods (see note)
net	Class library networking native code
nio	Class library NIO native code
wrappers	Class library VM interface native code

Note: When specifying the mt subcomponent you must also specify the method option.

<type> is the tracepoint type or group. The following types are supported:

- Entry
- Exit
- Event
- Exception
- Mem
- A group of tracepoints that have been specified by use of a group name. For example, nativeMethods select the group of tracepoints in MT (Method Trace) that relate to native methods. The following groups are supported:
 - compiledMethods
 - nativeMethods
 - staticMethods

<tracepoint_id>

is the tracepoint identifier. The tracepoint identifier constitutes the component name of the tracepoint, followed by its integer number inside that component. For example, j9mm.49, j9shr.20-29, j9vm.15. To understand these numbers, see “Determining the tracepoint ID of a tracepoint” on page 305.

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 tracepoints except j9vrb and j9trc:
-Xtrace:minimal={all,!j9vrb,!j9trc}
- All entry and exit tracepoints in j9bcu:
-Xtrace:maximal={j9bcu{entry},j9bcu{exit}}
- All tracepoints in j9mm except tracepoints 20-30:
-Xtrace:maximal=j9mm,maximal=!j9mm.20-30
- Tracepoints j9prt.5 through j9prt.15:

```
-Xtrace:print=tpnid{j9prt.5-15}
```

- All j9trc tracepoints:
-Xtrace:count=j9trc
- All entry and exit tracepoints:
-Xtrace:external={all{entry},all{exit}}
- All exception tracepoints:
-Xtrace:exception

or

```
-Xtrace:exception=all{exception}
```

- All exception tracepoints in j9bcu:
-Xtrace:exception=j9bcu
- Tracepoints j9prt.15 and j9shr.12:
-Xtrace:exception={j9prt.15,j9shr.12}

Trace levels:

Tracepoints have been assigned levels 0 through 9 that are based on the importance of the tracepoint.

A level 0 tracepoint is the most important. It is reserved for extraordinary events and errors. A level 9 tracepoint is in-depth component detail. To specify a given level of tracing, the level0 through level9 keywords are used. You can abbreviate these keywords to l0 through l9. For example, if level5 is selected, all tracepoints that have levels 0 through 5 are included. Level specifications do not apply to explicit tracepoint specifications that use the TPNID keyword.

The level is provided as a modifier to a component specification, for example:

```
-Xtrace:maximal={all{level5}}
```

or

```
-Xtrace:maximal={j9mm{l2},j9trc,j9bcu{level9},all{level1}}
```

In the first example, tracepoints that have a level of 5 or lower are enabled for all components. In the second example, all level 1 tracepoints are enabled. All level2 tracepoints in j9mm are enabled. All tracepoints up to level 9 are enabled in j9bcu.

Note: The level applies only to the current component. If multiple trace selection components are found in a trace properties file, the level is reset to the default for each new component.

Level specifications do not apply to explicit tracepoint specifications that use the TPNID keyword.

When the not operator is specified, the level is inverted; that is, !j9mm{level5} disables all tracepoints of level 6 or higher for the j9mm component. For example:

```
-Xtrace:print={all,!j9trc{l5},!j9mm{l6}}
```

enables trace for all components at level 9 (the default), but disables level 6 and higher for the locking component, and level 7 and higher for the storage component.

Examples

- Count the level zero and level one tracepoints matched:
`-Xtrace:count=all{L1}`
- Produce maximal trace of all components at level 5 and j9mm at level 9:
`-Xtrace:maximal={all{level5},j9mm{L9}}`
- Trace all components at level 6, but do not trace j9vrb at all, and do not trace the entry and exit tracepoints in the j9trc component:
`-Xtrace:minimal={all{L6},!j9vrb,!j9trc{entry},!j9trc{exit}}`

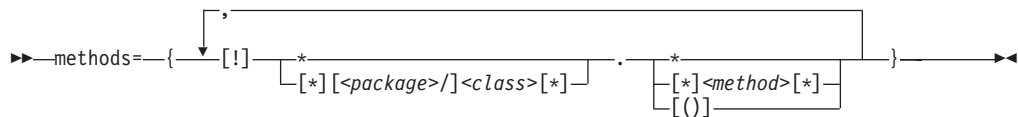
method=<method_specification>[,<method_specification>]

Using method trace provides a complete (and potentially large) diagnosis of code paths inside your application and the system classes. Use wild cards and filtering to control method trace so that you can focus on the sections of code that interest you.

Method trace can trace:

- Method entry
- Method exit

The methods parameter is defined as:



Where:

- The delimiter between parts of the package name is a forward slash, `"/"`.
- The `!` in the methods parameter is a NOT operator that allows you to tell the JVM not to trace the specified method or methods.
- The parentheses, `()`, define whether or not to include method parameters in the trace.
- If a method specification includes any commas, the whole specification must be enclosed in braces, for example:
`-Xtrace:methods={java/lang/*,java/util/*},print=mt`
- It might be necessary to enclose your command line in quotation marks to prevent the shell intercepting and fragmenting comma-separated command lines, for example:
`"-Xtrace:methods={java/lang/*,java/util/*},print=mt"`

To output all method trace information to stderr, use:

`-Xtrace:print=mt,methods=*. *`

Print method trace information for all methods to stderr.

`-Xtrace:iprint=mt,methods=*. *`

Print method trace information for all methods to stderr using indentation.

To output method trace information in binary format, see
"output=<filename>[,size[,<generations>]]" on page 298.

Examples

- **Tracing entry and exit of all methods in a given class:**

```
-Xtrace:methods={ReaderMain.*,java/lang/String.*},print=mt
```

This traces all method entry and exit of the ReaderMain class in the default package and the java.lang.String class.

- **Tracing entry, exit and input parameters of all methods in a class:**

```
-Xtrace:methods=ReaderMain.*(),print=mt
```

This traces all method entry, exit, and input of the ReaderMain class in the default package.

- **Tracing all methods in a given package:**

```
-Xtrace:methods=com/ibm/socket/*.*(),print=mt
```

This traces all method entry, exit, and input of all classes in the package com.ibm.socket.

- **Multiple method trace:**

```
-Xtrace:methods={Widget.*(),common/*},print=mt
```

This traces all method entry, exit, and input in the Widget class in the default package and all method entry and exit in the common package.

- **Using the ! operator**

```
-Xtrace:methods={ArticleUI.*,!ArticleUI.get*},print=mt
```

This traces all methods in the ArticleUI class in the default package except those beginning with “get”.

Example output

```
java "-Xtrace:methods={java/lang*.*},iprint=mt" HW
10:02:42.281*0x9e900 mt.4 > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
    V Compiled static method
10:02:42.281 0x9e900 mt.4 > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
    V Compiled static method
10:02:42.281 0x9e900 mt.4 > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
    V Compiled static method
10:02:42.281 0x9e900 mt.4 > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
    V Compiled static method
10:02:42.281 0x9e900 mt.10 < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
    V Compiled static method
10:02:42.281 0x9e900 mt.10 < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
    V Compiled static method
10:02:42.281 0x9e900 mt.4 > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
    V Compiled static method
10:02:42.281 0x9e900 mt.10 < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
    V Compiled static method
10:02:42.281 0x9e900 mt.10 < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
    V Compiled static method
10:02:42.281 0x9e900 mt.4 > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
    V Compiled static method
10:02:42.281 0x9e900 mt.4 > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
    V Compiled static method
10:02:42.296 0x9e900 mt.10 < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
    V Compiled static method
10:02:42.296 0x9e900 mt.10 < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
    V Compiled static method
10:02:42.296 0x9e900 mt.4 > java/lang/String.<clinit>()V Compiled static method
10:02:42.296 0x9e900 mt.4 > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
    V Compiled static method
10:02:42.296 0x9e900 mt.4 > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
```

```

V Compiled static method
10:02:42.296 0x9e900 mt.4      > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
V Compiled static method
10:02:42.296 0x9e900 mt.10     < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
V Compiled static method
10:02:42.296 0x9e900 mt.4      > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
V Compiled static method
10:02:42.328 0x9e900 mt.10     < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
V Compiled static method
10:02:42.328 0x9e900 mt.10     < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
V Compiled static method
10:02:42.328 0x9e900 mt.4      > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
V Compiled static method
10:02:42.328 0x9e900 mt.10     < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
V Compiled static method
10:02:42.328 0x9e900 mt.4      > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
V Compiled static method
10:02:42.328 0x9e900 mt.10     < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
V Compiled static method
10:02:42.328 0x9e900 mt.10     < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
V Compiled static method

```

The output lines comprise of:

- 0x9e900, the current **execenv** (execution environment). Because every JVM thread has its own **execenv**, you can regard **execenv** as a thread-id. All trace with the same **execenv** relates to a single thread.
- The individual tracepoint id in the mt component that collects and emits the data.
- The remaining fields show whether a method is being entered (>) or exited (<), followed by details of the method.

output=<filename>[,size[,<generations>]]

Use the output option to send trace data to <filename>. If the file does not already exist, it is created automatically. If it does already exist, it is overwritten.

Optionally:

- You can limit the file to *size* MB, at which point it wraps to the beginning. If you do not limit the file, it grows indefinitely, until limited by disk space.
- If you want the final trace filename to contain today's date, the PID number that produced the trace, or the time, do one of the following steps as appropriate (see also the examples at the end of this section).
 - To include today's date (in "yyyymmdd" format) in the trace filename, specify "%d" as part of the <filename>.
 - To include the pidnumber of the process that is generating the tracefile, specify "%p" as part of the <filename>.
 - To include the time (in 24-hour hhmmss format) in the trace filename, specify "%t" as part of the <filename>.
- You can specify generations as a value 2 through 36. These values cause up to 36 files to be used in a round-robin way when each file reaches its size threshold. When a file needs to be reused, it is overwritten. If generations is specified, the filename must contain a "#" (hash, pound symbol), which will be substituted with its generation identifier, the sequence of which is 0 through 9 followed by A through Z.

Note: When tracing to a file, buffers for each thread are written when the buffer is full or when the JVM terminates. If a thread has been inactive for a period of time before JVM termination, what seems to be 'old' trace data is written to the file.

When formatted, it then seems that trace data is missing from the other threads, but this is an unavoidable side-effect of the buffer-per-thread design. This effect becomes especially noticeable when you use the generation facility, and format individual earlier generations.

Examples

- Trace output goes to `/u/traces/gc.problem`; no size limit:
`-Xtrace:output=/u/traces/gc.problem,maximal=j9gc`
- Output goes to trace and will wrap at 2 MB:
`-Xtrace:output={trace,2m},maximal=j9gc`
- Output goes to `gc0.trc`, `gc1.trc`, `gc2.trc`, each 10 MB in size:
`-Xtrace:output={gc#.trc,10m,3},maximal=j9gc`
- Output filename contains today's date in `yyyymmdd` format (for example, `traceout.20041025.trc`):
`-Xtrace:output=traceout.%d.trc,maximal=j9gc`
- Output file contains the number of the process (the PID number) that generated it (for example, `tracefrompid2112.trc`):
`-Xtrace:output=tracefrompid%p.trc,maximal=j9gc`
- Output filename contains the time in `hhmmss` format (for example, `traceout.080312.trc`):
`-Xtrace:output=traceout.%t.trc,maximal=j9gc`

exception.output=<filename>[,nnnm]

Use the exception option to redirect exception trace data to `<filename>`.

If the file does not already exist, it is created automatically. If it does already exist, it is overwritten. Optionally, you can limit the file to `nnn` MB, at which point it wraps nondestructively to the beginning. If you do not limit the file, it grows indefinitely, until limited by disk space.

Optionally, if you want the final trace filename to contain today's date, the PID number that produced the trace, or the time, do one of the following steps as appropriate (see also the examples at the end of this section).

- To include today's date (in `"yyyymmdd"` format) in the trace filename, specify `"%d"` as part of the `<filename>`.
- To include the pidnumber of the process that is generating the tracefile, specify `"%p"` as part of the `<filename>`.
- To include the time (in 24-hour `hhmmss` format) in the trace filename, specify `"%t"` as part of the `<filename>`.

Examples

- Trace output goes to `/u/traces/exception.trc`. No size limit:
`-Xtrace:exception.output=/u/traces/exception.trc,maximal`
- Output goes to except and wraps at 2 MB:
`-Xtrace:exception.output={except,2m},maximal`
- Output filename contains today's date in `yyyymmdd` format (for example, `traceout.20041025.trc`):
`-Xtrace:exception.output=traceout.%d.trc,maximal`
- Output file contains the number of the process (the PID number) that generated it (for example, `tracefrompid2112.trc`):
`-Xtrace:exception.output=tracefrompid%p.trc,maximal`

- Output filename contains the time in hhmmss format (for example, `traceout.080312.trc`):
`-Xtrace:exception.output=traceout.%.t.trc,maximal`

resume

Resumes tracing globally.

Note that suspend and resume are not recursive. That is, two suspends that are followed by a single resume cause trace to be resumed.

Example

- Trace resumed (not much use as a startup option):
`-Xtrace:resume`

resumecount=<count>

This trace option determines whether tracing is enabled for each thread.

If <count> is greater than zero, each thread initially has its tracing disabled and must receive <count> **resumethis** actions before it starts tracing.

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

This system property is for use with the **trigger** property. For more information, see “`trigger=<clause>[,<clause>][,<clause>]...`” on page 301.

Example

- Start with all tracing turned off. Each thread starts tracing when it has had three **resumethis** actions performed on it:
`-Xtrace:resumecount=3`

stackdepth=<n>

Used to limit the amount of stack frame information collected.

Purpose

Use this option to limit the maximum number of stack frames reported by the `jstacktrace` trace trigger action. All stack frames are recorded by default.

Parameters

n Record *n* stack frames

suspend

Suspends tracing globally (for all threads and all forms of tracing) but leaves tracepoints activated.

Example

- Tracing suspended:
`-Xtrace:suspend`

suspendcount=<count>

This trace option determines whether tracing is enabled for each thread.

If *<count>* is greater than zero, each thread initially has its tracing enabled and must receive *<count>* suspend this action before it stops tracing.

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

This trace option is for use with the **trigger** option. For more information, see “trigger=<clause>[,<clause>][,<clause>]...”

Example

- Start with all tracing turned on. Each thread stops tracing when it has had three **suspendthis** actions performed on it:

```
-Xtrace:suspendcount=3
```

trigger=<clause>[,<clause>][,<clause>]...

This trace option determines when various triggered trace actions occur. Supported actions include turning tracing on and off for all threads, turning tracing on or off for the current thread, or producing various dumps.

This trace option does not control what is traced. It controls only whether the information that has been selected by the other trace options is produced as normal or is blocked.

Each clause of the **trigger** option can be **tpnid{...}**, **method{...}**, or **group{...}**. You can specify multiple clauses of the same type if required, but you do not need to specify all types. The clause types are as follows:

method{<methodspec>[,<entryAction>[,<exitAction>[,<delayCount>[,<matchcount>]]]]}

On entering a method that matches *<methodspec>*, the specified *<entryAction>* is run. On leaving a method that matches *<methodspec>*, the specified *<exitAction>* is run. If you specify a *<delayCount>*, the actions are performed only after a matching *<methodspec>* has been entered that many times. If you specify a *<matchCount>*, *<entryAction>* and *<exitAction>* are performed at most that many times.

group{<groupname>,<action>[,<delayCount>[,<matchcount>]]}

On finding any active tracepoint that is defined as being in trace group *<groupname>*, for example **Entry** or **Exit**, the specified action is run. If you specify a *<delayCount>*, the action is performed only after that many active tracepoints from group *<groupname>* have been found. If you specify a *<matchCount>*, *<action>* is performed at most that many times.

tpnid{<tpnid> | <tpnidRange>,<action>[,<delayCount>[,<matchcount>]]}

On finding the specified active *<tpnid>* (tracepoint ID) or a *<tpnid>* that falls inside the specified *<tpnidRange>*, the specified action is run. If you specify a *<delayCount>*, the action is performed only after the JVM finds such an active *<tpnid>* that many times. If you specify a *<matchCount>*, *<action>* is performed at most that many times.

Actions

Wherever an action must be specified, you must select from these choices:

abort

Halt the JVM.

coredump

See **sysdump**.

heapdump

Produce a Heapdump. See Chapter 23, “Using Heapdump,” on page 257.

javadump

Produce a Javadump. See Chapter 22, “Using Javadump,” on page 245.

jstacktrace

Examine the Java stack of the current thread and generate auxiliary tracepoints for each stack frame. The auxiliary tracepoints are written to the same destination as the tracepoint or method trace that triggered the action. You can control the number of stack frames examined with the **stackdepth=n** option. See “stackdepth=<n>” on page 300. The **jstacktrace** action is available from Java 5 SR 10.

resume

Resume all tracing (except for threads that are suspended by the action of the **resumecount** property and `Trace.suspendThis()` calls).

resumethis

Decrement the suspend count for this thread. If the suspend count is zero or below, resume tracing for this thread.

segv

Cause a segmentation violation. (Intended for use in debugging.)

snap

Snap all active trace buffers to a file in the current working directory. The file name has the format: `Snapnnnn.yyyymmdd.hhmmssst.ppppp.trc`, where `nnnn` is the sequence number of the snap file since JVM startup, `yyymmdd` is the date, `hhmmssst` 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 Chapter 24, “Using system dumps and the dump viewer,” on page 263.

Examples

- To start tracing this thread when it enters any method in `java/lang/String`, and to stop tracing the thread after exiting the method:

```
-Xtrace:resumecount=1
-Xtrace:trigger=method{java/lang/String.*,resumethis,suspendthis}
```
- To resume all tracing when any thread enters a method in any class that starts with “error”:

```
-Xtrace:trigger=method{*.error*,resume}
```
- To produce a core dump when you reach the 1000th and 1001st tracepoint from the “jvmri” trace group.

Note: Without `<matchcount>`, you risk filling your disk with coredump files.

```
-Xtrace:trigger=group{staticmethods,coredump,1000,2}
```

If using the **trigger** option generates multiple dumps in rapid succession (more than one per second), specify a dump option to guarantee unique dump names. See Chapter 21, “Using dump agents,” on page 223 for more information.

- To trace (all threads) while the application is active; that is, not starting or shut down. (The application name is "HelloWorld"):
-Xtrace:suspend,trigger=method{HelloWorld.main,resume,suspend}
- To print a Java stack trace to the console when the mycomponent.1 tracepoint is reached:
-Xtrace:print=mycomponent.1,trigger=tpnid{mycomponent.1,jstacktrace}
- To write a Java stack trace to the trace output file when the Sample.code() method is called:
-Xtrace:maximal=mt,output=trc.out,methods={mycompany/mypackage/Sample.code},trigger=method{mycom

Using the Java API

You can dynamically control trace in a number of ways from a Java application by using the `com.ibm.jvm.Trace` class.

Activating and deactivating tracepoints

```
int set(String cmd);
```

The `Trace.set()` method allows a Java application to select tracepoints dynamically. For example:

```
Trace.set("iprint=all");
```

The syntax is the same as that used in a trace properties file for the `print`, `iprint`, `count`, `maximal`, `minimal` and `external` trace options.

A single trace command is parsed per invocation of `Trace.set`, so to achieve the equivalent of **-Xtrace:maximal=j9mm,iprint=j9shr** two calls to `Trace.set` are needed with the parameters `maximal=j9mm` and `iprint=j9shr`

Obtaining snapshots of trace buffers

```
void snap();
```

You must have activated trace previously with the **maximal** or **minimal** options and without the **out** option.

Suspending or resuming trace

```
void suspend();
```

The `Trace.suspend()` method suspends tracing for all the threads in the JVM.

```
void resume();
```

The `Trace.resume()` method resumes tracing for all threads in the JVM. It is not recursive.

```
void suspendThis();
```

The `Trace.suspendThis()` method decrements the suspend and resume count for the current thread and suspends tracing the thread if the result is negative.

```
void resumeThis();
```

The `Trace.resumeThis()` method increments the suspend and resume count for the current thread and resumes tracing the thread if the result is not negative.

Using the trace formatter

The trace formatter is a Java program that converts binary trace point data in a trace file to a readable form. The formatter requires the `J9TraceFormat.dat` file, which contains the formatting templates. The formatter produces a file containing header information about the JVM that produced the binary trace file, a list of threads for which trace points were produced, and the formatted trace points with their timestamp, thread ID, trace point ID and trace point data.

To use the trace formatter on a binary trace file type:

```
java com.ibm.jvm.format.TraceFormat <input_file> [<output_file>] [options]
```

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

If you do not specify an output file, the output file is called *<input_file>.fmt*.

The size of the heap needed to format the trace is directly proportional to the number of threads present in the trace file. For large numbers of threads the formatter might run out of memory, generating the error `OutOfMemoryError`. In this case, increase the heap size using the `-Xmx` option.

Available options

The following options are available with the trace formatter:

-datdir *<directory>*

Selects an alternative formatting template file directory. The directory must contain the `J9TraceFormat.dat` file.

-help

Displays usage information.

-indent

Indents trace messages at each Entry trace point and outdents trace messages at each Exit trace point. The default is not to indent the messages.

-overridezone *<hours>*

Add *<hours>* hours to formatted tracepoints, the value can be negative. This option allows the user to override the default time zone used in the formatter (UTC).

-summary

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

-thread:*<thread id>*[,*<thread id>*]...

Filters the output for the given thread IDs only. *thread id* is the ID of the thread, which can be specified in decimal or hex (0x) format. Any number of thread IDs can be specified, separated by commas.

-uservmid *<string>*

Inserts *<string>* in each formatted tracepoint. The string aids reading or parsing when several different JVMs or JVM runs are traced for comparison.

Determining the tracepoint ID of a tracepoint

Throughout the code that makes up the JVM, there are numerous tracepoints. Each tracepoint maps to a unique id consisting of the name of the component containing the tracepoint, followed by a period (".") and then the numeric identifier of the tracepoint.

These tracepoints are also recorded in two .dat files (TraceFormat.dat and J9TraceFormat.dat) that are shipped with the JRE and the trace formatter uses these files to convert compressed trace points into readable form.

JVM developers and Service can use the two .dat files to enable formulation of trace point ids and ranges for use under **-Xtrace** when tracking down problems. The next sample taken from the top of TraceFormat.dat, which illustrates how this mechanism works:

```
5.0
j9bcu 0 1 1 N Trc_BCU_VMInitStages_Event1 " Trace engine initialized for module j9dyn"
j9bcu 2 1 1 N Trc_BCU_internalDefineClass_Entry " >internalDefineClass %p"
j9bcu 4 1 1 N Trc_BCU_internalDefineClass_Exit " <internalDefineClass %p ->"
j9bcu 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> <t> <o> <l> <e> <symbol> <template>`

where:

`<component>`

is the SDK component name.

`<t>` is the tracepoint type (0 through 12), where these types are used:

- 0 = event
- 1 = exception
- 2 = function entry
- 4 = function exit
- 5 = function exit with exception
- 8 = internal
- 12 = assert

`<o>` is the overhead (0 through 10), which determines whether the tracepoint is compiled into the runtime JVM code.

`<l>` is the level of the tracepoint (0 through 9). High frequency tracepoints, known as hot tracepoints, are assigned higher level numbers.

`<e>` is an internal flag (Y/N) and no longer used.

`<symbol>`

is the internal symbolic name of the tracepoint.

`<template>`

is a template in double quotation marks that is used to format the entry.

For example, if you discover that a problem occurred somewhere close to the issue of Trc_BCU_VMInitStages_Event, you can rerun the application with **-Xtrace:print=tpnid{j9bcu.0}**. That command will result in an output such as:

```
14:10:42.717*0x41508a00    j9bcu.0          - Trace engine initialized for module j9dyn
```

The example given is fairly trivial. However, the use of tpnid ranges and the formatted parameters contained in most trace entries provides a very powerful problem debugging mechanism.

The .dat files contain a list of all the tracepoints ordered by component, then sequentially numbered from 0. The full tracepoint id is included in all formatted output of a tracepoint; For example, tracing to the console or formatted binary trace.

The format of trace entries and the contents of the .dat files are subject to change without notice. However, the version number should guarantee a particular format.

Application trace

Application trace allows you to trace Java applications using the JVM trace facility.

You must register your Java application with application trace and add trace calls where appropriate. After you have started an application trace module, you can enable or disable individual tracepoints at any time.

Implementing application trace

Application trace is in the package `com.ibm.jvm.Trace`. The application trace API is described in this section.

Registering for trace

Use the `registerApplication()` method to specify the application to register with application trace.

The method is of the form:

```
int registerApplication(String application_name, String[] format_template)
```

The `application_name` argument is the name of the application you want to trace. The name must be the same as the application name you specify at JVM startup. The `format_template` argument is an array of format strings like the strings used by the `printf` method. You can specify templates of up to 16 KB. The position in the array determines the tracepoint identifier (starting at 0). You can use these identifiers to enable specific tracepoints at run time. The first character of each template is a digit that identifies the type of tracepoint. The tracepoint type can be one of entry, exit, event, exception, or exception exit. After the tracepoint type character, the template has a blank character, followed by the format string.

The trace types are defined as static values within the `Trace` class:

```
public static final String EVENT= "0 ";
public static final String EXCEPTION= "1 ";
public static final String ENTRY= "2 ";
public static final String EXIT= "4 ";
public static final String EXCEPTION_EXIT= "5 ";
```

The `registerApplication()` method returns an integer value. Use this value in subsequent `trace()` calls. If the `registerApplication()` method call fails for any reason, the value returned is -1.

Tracepoints

These trace methods are implemented.

```

void trace(int handle, int traceId);
void trace(int handle, int traceId, String s1);
void trace(int handle, int traceId, String s1, String s2);
void trace(int handle, int traceId, String s1, String s2, String s3);
void trace(int handle, int traceId, String s1, Object o1);
void trace(int handle, int traceId, Object o1, String s1);
void trace(int handle, int traceId, String s1, int i1);
void trace(int handle, int traceId, int i1, String s1);
void trace(int handle, int traceId, String s1, long l1);
void trace(int handle, int traceId, long l1, String s1);
void trace(int handle, int traceId, String s1, byte b1);
void trace(int handle, int traceId, byte b1, String s1);
void trace(int handle, int traceId, String s1, char c1);
void trace(int handle, int traceId, char c1, String s1);
void trace(int handle, int traceId, String s1, float f1);
void trace(int handle, int traceId, float f1, String s1);
void trace(int handle, int traceId, String s1, double d1);
void trace(int handle, int traceId, double d1, String s1);
void trace(int handle, int traceId, Object o1);
void trace(int handle, int traceId, Object o1, Object o2);
void trace(int handle, int traceId, int i1);
void trace(int handle, int traceId, int i1, int i2);
void trace(int handle, int traceId, int i1, int i2, int i3);
void trace(int handle, int traceId, long l1);
void trace(int handle, int traceId, long l1, long l2);
void trace(int handle, int traceId, long l1, long l2, long i3);
void trace(int handle, int traceId, byte b1);
void trace(int handle, int traceId, byte b1, byte b2);
void trace(int handle, int traceId, byte b1, byte b2, byte b3);
void trace(int handle, int traceId, char c1);
void trace(int handle, int traceId, char c1, char c2);
void trace(int handle, int traceId, char c1, char c2, char c3);
void trace(int handle, int traceId, float f1);
void trace(int handle, int traceId, float f1, float f2);
void trace(int handle, int traceId, float f1, float f2, float f3);
void trace(int handle, int traceId, double d1);
void trace(int handle, int traceId, double d1, double d2);
void trace(int handle, int traceId, double d1, double d2, double d3);
void trace(int handle, int traceId, String s1, Object o1, String s2);
void trace(int handle, int traceId, Object o1, String s1, Object o2);
void trace(int handle, int traceId, String s1, int i1, String s2);
void trace(int handle, int traceId, int i1, String s1, int i2);
void trace(int handle, int traceId, String s1, long l1, String s2);
void trace(int handle, int traceId, long l1, String s1, long l2);
void trace(int handle, int traceId, String s1, byte b1, String s2);
void trace(int handle, int traceId, byte b1, String s1, byte b2);
void trace(int handle, int traceId, String s1, char c1, String s2);
void trace(int handle, int traceId, char c1, String s1, char c2);
void trace(int handle, int traceId, String s1, float f1, String s2);
void trace(int handle, int traceId, float f1, String s1, float f2);
void trace(int handle, int traceId, String s1, double d1, String s2);
void trace(int handle, int traceId, double d1, String s1, double d2);

```

The handle argument is the value returned by the registerApplication() method. The traceId argument is the number of the template entry starting at 0.

Printf specifiers

Application trace supports the ANSI C printf specifiers. You must be careful when you select the specifier; otherwise you might get unpredictable results, including abnormal termination of the JVM.

For 64-bit integers, you must use the ll (lower case LL, meaning long long) modifier. For example: %lld or %lli.

For pointer-sized integers use the z modifier. For example: %zx or %zd.

Example HelloWorld with application trace

This code illustrates a “HelloWorld” application with application trace.

For more information about this example, see “Using application trace at run time.”

```
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 308 uses the `Trace.set()` API to pass arguments to the trace function. For example, to pass the **iprint** argument to the trace function, use the following command:

```
java HelloWorld iprint=HelloWorld
```

Starting the example HelloWorld application in this way produces the following results:

```
Trace setting: iprint=HelloWorld
09:50:29.417*0x2a08a00 084002 - Event id 1, text = Trace initialized
09:50:29.417 0x2a08a00 084000 > Entering sayHello
Hello
09:50:29.427 0x2a08a00 084001 < Exiting sayHello
09:50:29.427 0x2a08a00 084000 > Entering sayGoodbye
Bye
09:50:29.437 0x2a08a00 084004 * < Exception exit from sayGoodbye
```

You can also specify trace options directly by using the **-Xtrace** option. See “Options that control tracepoint activation” on page 291 for more details. For example, you can obtain a similar result to the previous command by using the **-Xtrace** option to specify **iprint** on the command line:

```
java -Xtrace:iprint=HelloWorld HelloWorld
```

Note: You can enable tracepoints by application name and by tracepoint number. Using tracepoint “levels” or “types” is not supported for application trace.

Using method trace

Method trace is a powerful tool for tracing methods in any Java code.

Method trace provides a comprehensive and detailed diagnosis of code paths inside your application, and also inside the system classes. You do not have to add any hooks or calls to existing code. You can focus on interesting code by using wild cards and filtering to control method trace.

Method trace can trace:

- Method entry
- Method exit

Use method trace to debug and trace application code and the system classes provided with the JVM.

While method trace is powerful, it also has a cost. Application throughput is affected by method trace. The impact is proportion to the number of methods traced. Additionally, trace output is reasonably large and might require a large amount of drive space. For instance, a full method trace of a “Hello World” application is over 10 MB.

Running with method trace

Control method trace by using the command-line option **-Xtrace:<option>**.

To produce method trace you need to set trace options for the Java classes and methods you want to trace. You also need to route the method trace to the destination you require.

You must set the following two options:

1. Use **-Xtrace:methods** to select which Java classes and methods you want to trace.
2. Use either
 - **-Xtrace:print** to route the trace to stderr.
 - **-Xtrace:maximal** and **-Xtrace:output** to route the trace to a binary compressed file using memory buffers.

Use the **methods** parameter to control what is traced. For example, to trace all methods on the String class, set **-Xtrace:methods=java/lang/String.*,print=mt**.

The **methods** parameter is formally defined as follows:

```
-Xtrace:methods=[(!]<method_spec>[,...]]
```

Where *<method_spec>* is formally defined as:

```
{*|[*]<classname>[*]}.{|[*]<methodname>[*]}[()]
```

Note:

- The delimiter between parts of the package name is a forward slash, '/', even on Windows platforms where a backward slash is a path delimiter.
- The symbol "!" in the methods parameter is a NOT operator. Use this symbol to exclude methods from the trace. Use "this" with other **methods** parameters to set up a trace of the form: "trace methods of this type but not methods of that type".
- The parentheses, (), that are in the *<method_spec>* define whether to trace method parameters.
- If a method specification includes any commas, the whole specification must be enclosed in braces:

```
-Xtrace:methods={java/lang/*,java/util/*},print=mt
```
- On Linux, AIX, z/OS, and i5/OS, you might have to enclose your command line in quotation marks. This action prevents the shell intercepting and fragmenting comma-separated command lines:

```
"-Xtrace:methods={java/lang/*,java/util/*},print=mt"
```

Use the **print**, **maximal** and **output** options to route the trace to the required destination, where:

- **print** formats the tracepoint data while the Java application is running and writes the tracepoints to stderr.
- **maximal** saves the tracepoints into memory buffers.
- **output** writes the memory buffers to a file, in a binary compressed format.

To produce method trace that is routed to stderr, use the **print** option, specifying **mt** (method trace). For example: **-Xtrace:methods=java/lang/String.*,print=mt**.

To produce method trace that is written to a binary file from the memory buffers, use the **maximal** and **output** options. For example: **-Xtrace:methods=java/lang/String.*,maximal=mt,output=mytrace.trc**.

If you want your trace output to contain only the tracepoints you specify, use the option **-Xtrace:none** to switch off the default tracepoints. For example: **java -Xtrace:none -Xtrace:methods=java/lang/String.*,maximal=mt,output=mytrace.trc <class>**.

Untraceable methods

Internal Native Library (INL) native methods inside the JVM cannot be traced because they are not implemented using JNI. The list of methods that are not traceable is subject to change without notice between releases.

The INL native methods in the JVM include:

```
java.lang.Class.allocateAndFillArray
java.lang.Class.forNameImpl
java.lang.Class.getClassDepth
java.lang.Class.getClassLoaderImpl
java.lang.Class.getComponentType
java.lang.Class.getConstructorImpl
java.lang.Class.getConstructorsImpl
java.lang.Class.getDeclaredClassesImpl
java.lang.Class.getDeclaredConstructorImpl
java.lang.Class.getDeclaredConstructorsImpl
java.lang.Class.getDeclaredFieldImpl
java.lang.Class.getDeclaredFieldsImpl
java.lang.Class.getDeclaredMethodImpl
java.lang.Class.getDeclaredMethodsImpl
java.lang.Class.getDeclaringClassImpl
java.lang.Class.getEnclosingObject
java.lang.Class.getEnclosingObjectClass
java.lang.Class.getFieldImpl
java.lang.Class.getFieldsImpl
java.lang.Class.getGenericSignature
java.lang.Class.getInterfaceMethodCountImpl
java.lang.Class.getInterfaceMethodsImpl
java.lang.Class.getInterfaces
java.lang.Class.getMethodImpl
java.lang.Class.getModifiersImpl
java.lang.Class.getNameImpl
java.lang.Class.getSimpleNameImpl
java.lang.Class.getStackClass
java.lang.Class.getStackClasses
java.lang.Class.getStaticMethodCountImpl
java.lang.Class.getStaticMethodsImpl
java.lang.Class.getSuperclass
java.lang.Class.getVirtualMethodCountImpl
java.lang.Class.getVirtualMethodsImpl
java.lang.Class.isArray
java.lang.Class.isAssignableFrom
java.lang.Class.isInstance
java.lang.Class.isPrimitive
java.lang.Class.newInstanceImpl
java.lang.ClassLoader.findLoadedClassImpl
java.lang.ClassLoader.getStackClassLoader
java.lang.ClassLoader.loadLibraryWithPath
java.lang.J9VMInternals.getInitStatus
java.lang.J9VMInternals.getInitThread
java.lang.J9VMInternals.initializeImpl
java.lang.J9VMInternals.sendClassPrepareEvent
java.lang.J9VMInternals.setInitStatusImpl
java.lang.J9VMInternals.setInitThread
java.lang.J9VMInternals.verifyImpl
java.lang.J9VMInternals.getStackTrace
java.lang.Object.clone
java.lang.Object.getClass
java.lang.Object.hashCode
java.lang.Object.notify
java.lang.Object.notifyAll
java.lang.Object.wait
java.lang.ref.Finalizer.runAllFinalizersImpl
java.lang.ref.Finalizer.runFinalizationImpl
java.lang.ref.Reference.getImpl
```



```

java.lang.ref.Reference.initReferenceImpl
java.lang.reflect.AccessibleObject.checkAccessibility
java.lang.reflect.AccessibleObject.getAccessibleImpl
java.lang.reflect.AccessibleObject.getExceptionTypesImpl
java.lang.reflect.AccessibleObject.getModifiersImpl
java.lang.reflect.AccessibleObject.getParameterTypesImpl
java.lang.reflect.AccessibleObject.getSignature
java.lang.reflect.AccessibleObject.getStackClass
java.lang.reflect.AccessibleObject.initializeClass
java.lang.reflect.AccessibleObject.invokeImpl
java.lang.reflect.AccessibleObject.setAccessibleImpl
java.lang.reflect.Array.get
java.lang.reflect.Array.getBoolean
java.lang.reflect.Array.getByte
java.lang.reflect.Array.getChar
java.lang.reflect.Array.getDouble
java.lang.reflect.Array.getFloat
java.lang.reflect.Array.getInt
java.lang.reflect.Array.getLength
java.lang.reflect.Array.getLong
java.lang.reflect.Array.getShort
java.lang.reflect.Array.multiNewArrayImpl
java.lang.reflect.Array.newArrayImpl
java.lang.reflect.Array.set
java.lang.reflect.Array.setBoolean
java.lang.reflect.Array.setByte
java.lang.reflect.Array.setChar
java.lang.reflect.Array.setDouble
java.lang.reflect.Array.setFloat
java.lang.reflect.Array.setImpl
java.lang.reflect.Array.setInt
java.lang.reflect.Array.setLong
java.lang.reflect.Array.setShort
java.lang.reflect.Constructor.newInstanceImpl
java.lang.reflect.Field.getBooleanImpl
java.lang.reflect.Field.getByteImpl
java.lang.reflect.Field.getCharImpl
java.lang.reflect.Field.getDoubleImpl
java.lang.reflect.Field.getFloatImpl
java.lang.reflect.Field.getImpl
java.lang.reflect.Field.getIntImpl
java.lang.reflect.Field.getLongImpl
java.lang.reflect.Field.getModifiersImpl
java.lang.reflect.Field.getNameImpl
java.lang.reflect.Field.getShortImpl
java.lang.reflect.Field.getSignature
java.lang.reflect.Field.getTypeImpl
java.lang.reflect.Field.setBooleanImpl
java.lang.reflect.Field.setByteImpl
java.lang.reflect.Field.setCharImpl
java.lang.reflect.Field.setDoubleImpl
java.lang.reflect.Field.setFloatImpl
java.lang.reflect.Field.setImpl
java.lang.reflect.Field.setIntImpl
java.lang.reflect.Field.setLongImpl
java.lang.reflect.Field.setShortImpl
java.lang.reflect.Method.getNameImpl
java.lang.reflect.Method.getReturnTypeImpl
java.lang.String.intern
java.lang.String.isResettableJVM0
java.lang.System.arraycopy
java.lang.System.currentTimeMillis
java.lang.System.hiresClockImpl
java.lang.System.hiresFrequencyImpl
java.lang.System.identityHashCode
java.lang.System.nanoTime
java.lang.Thread.currentThread

```



```

java.lang.Thread.getStackTraceImpl
java.lang.Thread.holdsLock
java.lang.Thread.interrupted
java.lang.Thread.interruptImpl
java.lang.Thread.isInterruptedImpl
java.lang.Thread.resumeImpl
java.lang.Thread.sleep
java.lang.Thread.startImpl
java.lang.Thread.stopImpl
java.lang.Thread.suspendImpl
java.lang.Thread.yield
java.lang.Throwable.fillInStackTrace
java.security.AccessController.getAccessControlContext
java.security.AccessController.getProtectionDomains
java.security.AccessController.getProtectionDomainsImpl
org.apache.harmony.kernel.vm.VM.getStackClassLoader
org.apache.harmony.kernel.vm.VM.internImpl

```

Examples of use

Here are some examples of method trace commands and their results.

- **Tracing entry and exit of all methods in a given class:**

```
-Xtrace:methods=java/lang/String.*,print=mt
```

This example traces entry and exit of all methods in the `java.lang.String` class.

The name of the class must include the full package name, using '/' as a separator. The method name is separated from the class name by a dot '.' In this example, '*' is used to include all methods. Sample output:

```

09:39:05.569 0x1a1100 mt.0 > java/lang/String.length()I Bytecode method, This = 8b27d8
09:39:05.579 0x1a1100 mt.6 < java/lang/String.length()I Bytecode method

```

- **Tracing method input parameters:**

```
-Xtrace:methods=java/lang/Thread.*(),print=mt
```

This example traces all methods in the `java.lang.Thread` class, with the parentheses '()' indicating that the trace should also include the method call parameters. The output includes an extra line, giving the class and location of the object on which the method was called, and the values of the parameters. In this example the method call is `Thread.join(long millis,int nanos)`, which has two parameters:

```

09:58:12.949 0x4236ce00 mt.0 > java/lang/Thread.join(JI)V Bytecode method, This = 8ffd20
09:58:12.959 0x4236ce00 mt.18 - Instance method receiver: com/ibm/tools/attach/javaSE/AttachHan
arguments: ((long)1000,(int)0)

```

- **Tracing multiple methods:**

```
-Xtrace:methods={java/util/HashMap.size,java/lang/String.length},print=mt
```

This example traces the `size` method on the `java.util.HashMap` class and the `length` method on the `java.lang.String` class. The method specification includes the two methods separated by a comma, with the entire method specification enclosed in braces '{' and '}'. Sample output:

```

10:28:19.296 0x1a1100 mt.0 > java/lang/String.length()I Bytecode method, This = 8c2548
10:28:19.306 0x1a1100 mt.6 < java/lang/String.length()I Bytecode method
10:28:19.316 0x1a1100 mt.0 > java/util/HashMap.size()I Bytecode method, This = 8dd7e8
10:28:19.326 0x1a1100 mt.6 < java/util/HashMap.size()I Bytecode method

```

- **Using the ! (not) operator to select tracepoints:**

```
-Xtrace:methods={java/util/HashMap.*,!java/util/HashMap.put*},print
```

This example traces all methods in the `java.util.HashMap` class except those beginning with `put`. Sample output:

```

10:37:42.225 0x1a1100 mt.0 > java/util/HashMap.createHashedEntry(Ljava/lang/Object;II)Ljava/util/H
e0
10:37:42.246 0x1a1100 mt.6 < java/util/HashMap.createHashedEntry(Ljava/lang/Object;II)Ljava/util/H
10:37:42.256 0x1a1100 mt.1 > java/util/HashMap.findNonNullKeyEntry(Ljava/lang/Object;II)Ljava/util
d7e0
10:37:42.266 0x1a1100 mt.7 < java/util/HashMap.findNonNullKeyEntry(Ljava/lang/Object;II)Ljava/util

```

Example of method trace output

An example of method trace output.

Sample output using the command **java -Xtrace:iprint=mt,methods=java/lang/**
***,* -version:**

```

10:02:42.281*0x9e900      mt.4      > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                          V Compiled static method
10:02:42.281 0x9e900      mt.4      > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                          V Compiled static method
10:02:42.281 0x9e900      mt.4      > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                          V Compiled static method
10:02:42.281 0x9e900      mt.4      > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                          V Compiled static method
10:02:42.281 0x9e900      mt.10     < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                          V Compiled static method
10:02:42.281 0x9e900      mt.10     < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                          V Compiled static method
10:02:42.281 0x9e900      mt.4      > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                          V Compiled static method
10:02:42.281 0x9e900      mt.10     < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                          V Compiled static method
10:02:42.281 0x9e900      mt.10     < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                          V Compiled static method
10:02:42.281 0x9e900      mt.4      > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                          V Compiled static method
10:02:42.281 0x9e900      mt.4      > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                          V Compiled static method
10:02:42.296 0x9e900      mt.10     < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                          V Compiled static method
10:02:42.296 0x9e900      mt.10     < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                          V Compiled static method
10:02:42.296 0x9e900      mt.4      > java/lang/String.<clinit>()V Compiled static method
10:02:42.296 0x9e900      mt.4      > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                          V Compiled static method
10:02:42.296 0x9e900      mt.4      > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                          V Compiled static method
10:02:42.296 0x9e900      mt.4      > java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                          V Compiled static method
10:02:42.296 0x9e900      mt.10     < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                          V Compiled static method
10:02:42.296 0x9e900      mt.4      > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                          V Compiled static method
10:02:42.328 0x9e900      mt.10     < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                          V Compiled static method
10:02:42.328 0x9e900      mt.10     < java/lang/J9VMInternals.verify(Ljava/lang/Class;)
                          V Compiled static method
10:02:42.328 0x9e900      mt.4      > java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                          V Compiled static method
10:02:42.328 0x9e900      mt.10     < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                          V Compiled static method
10:02:42.328 0x9e900      mt.4      > java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                          V Compiled static method
10:02:42.328 0x9e900      mt.10     < java/lang/J9VMInternals.setInitStatus(Ljava/lang/Class;I)
                          V Compiled static method
10:02:42.328 0x9e900      mt.10     < java/lang/J9VMInternals.initialize(Ljava/lang/Class;)
                          V Compiled static method

```

The output lines comprise:

- 0x9e900, the current **execenv** (execution environment). Because every JVM thread has its own **execenv**, you can regard **execenv** as a thread-id. All trace with the same **execenv** relates to a single thread.
- The individual tracepoint id in the mt component that collects and emits the data.
- The remaining fields show whether a method is being entered (>) or exited (<), followed by details of the method.

Chapter 26. JIT problem determination

You can use command-line options to help diagnose JIT compiler problems and to tune performance.

- “Disabling the JIT compiler”
- “Selectively disabling the JIT compiler” on page 318
- “Locating the failing method” on page 319
- “Identifying JIT compilation failures” on page 320
- “Performance of short-running applications” on page 321
- “JVM behavior during idle periods” on page 321

Diagnosing a JIT problem

Occasionally, valid bytecodes might compile into invalid native code, causing the Java program to fail. By determining whether the JIT 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 compiler

If you suspect that a problem is occurring in the JIT 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. 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 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 compiler disabled).

Procedure

1. Remove any **-Xjit** options (and accompanying parameters) from your command line.

2. Use the **-Xint** command-line option to disable the JIT compiler. For performance reasons, do not use the **-Xint** option in a production environment.

What to do next

Running the Java program with the compilation disabled leads to one of the following:

- The failure remains. The problem is not in the JIT 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 compiler.

Selectively disabling the JIT compiler

If your Java program failure points to a problem with the JIT compiler, you can try to narrow down the problem further.

About this task

By default, the JIT compiler optimizes methods at various optimization levels. Different selections of optimizations are applied to different methods, based on their call counts. Methods that are called more frequently are optimized at higher levels. By changing JIT compiler parameters, you can control the optimization level at which methods are optimized. You can determine whether the optimizer is at fault and, if it is, which optimization is problematic.

You specify JIT parameters as a comma-separated list, appended to the **-Xjit** option. The syntax is **-Xjit:<param1>,<param2>=<value>**. For example:

```
java -Xjit:verbose,optLevel=noOpt HelloWorld
```

runs the HelloWorld program, enables verbose output from the JIT, and makes the JIT generate native code without performing any optimizations. Optimization options are listed in “How the JIT compiler optimizes code” on page 36.

Follow these steps to determine which part of the compiler is causing the failure:

Procedure

1. Set the JIT parameter **count=0** to change the compilation threshold to zero. This parameter causes each Java method to be compiled before it is run. Use **count=0** only when diagnosing problems, because a lot more methods are compiled, including methods that are used infrequently. The extra compilation uses more computing resources and slows down your application. With **count=0**, your application fails immediately when the problem area is reached. In some cases, using **count=1** can reproduce the failure more reliably.
2. Add **disableInlining** to the JIT 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. 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 parameter list does not cause the failure to reappear, do so to improve performance. Follow the instructions in “Locating the failing method” to improve the performance of the workaround.

If the failure still occurs at the “noOpt” optimization level, you must disable the JIT compiler as a workaround.

Locating the failing method

When you have determined the lowest optimization level at which the JIT compiler must compile methods to trigger the failure, you can find out which part of the Java program, when compiled, causes the failure. You can then instruct the compiler to limit the workaround to a specific method, class, or package, allowing the compiler to compile the rest of the program as usual. For JIT compiler failures, if the failure occurs with **-Xjit:optLevel=noOpt**, you can also instruct the compiler to not compile the method or methods that are causing the failure at all.

Before you begin

If you see error output like this example, you can use it to identify the failing method:

```
Unhandled exception
Type=Segmentation error vmState=0x00000000
Target=2_30_20050520_01866_BHdSMr (Linux 2.4.21-27.0.2.EL)
CPU=s390x (2 logical CPUs) (0x7b6a8000 RAM)
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=4148bf20 Signal_Code=00000001
Handler1=00000100002ADB14 Handler2=00000100002F480C InaccessibleAddress=0000000000000000
gpr0=0000000000000006 gpr1=0000000000000006 gpr2=0000000000000000 gpr3=0000000000000006
gpr4=0000000000000001 gpr5=0000000080056808 gpr6=0000010002BCCA20 gpr7=0000000000000000
.....
Compiled_method=java/security/AccessController.toArrayOfProtectionDomains([Ljava/lang/Object;
Ljava/security/AccessControlContext;)[Ljava/security/ProtectionDomain;
```

The important lines are:

vmState=0x00000000

Indicates that the code that failed was not JVM runtime code.

Module= or Module_base_address=

Not in the output (might be blank or zero) because the code was compiled by the JIT, and outside any DLL or library.

Compiled_method=

Indicates the Java method for which the compiled code was produced.

About this task

If your output does not indicate the failing method, follow these steps to identify the failing method:

Procedure

1. Run the Java program with the JIT parameters **verbose** and **vlog=<filename>** added to the **-Xjit** option. With these parameters, the compiler lists compiled methods in a log file named *<filename>.<date>.<time>.<pid>*, also called a *limit file*. A typical limit file contains lines that correspond to compiled methods, like:

```
+ (hot) java/lang/Math.max(II)I @ 0x10C11DA4-0x10C11DDD
```

Lines that do not start with the plus sign are ignored by the compiler in the following steps and you can remove them from the file.

2. Run the program again with the JIT 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. 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. Repeat this process using different values for *<m>* and *<n>*, as many times as necessary, to find the minimum set of methods that must be compiled to trigger the failure. By halving the number of selected lines each time, you can perform a binary search for the failing method. Often, you can reduce the file to a single line.

What to do next

When you have located the failing method, you can disable the JIT 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}
```

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 (`libj9jit23.so`, or `j9jit23.dll` on Windows), 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=00000007FE05645B8 Handler2=00000007FE0615C20
```



```

R0=E8D4001870C00001 R1=0000007FF49181E0 R2=0000007FE2FBCEE0 R3=0000007FF4E60D70
R4=E8D4001870C00000 R5=0000007FE2E02D30 R6=0000007FF4C0F188 R7=0000007FE2F8C290
.....
Module=/home/test/sdk/jre/bin/libj9jit23.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 248

Module=/home/test/sdk/jre/bin/libj9jit23.so

Indicates that the error occurred in libj9jit23.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 319). If excluding the method prevents the crash, you have a workaround that you can use while the service team corrects your problem.

Performance of short-running applications

The IBM JIT compiler is tuned for long-running applications typically used on a server. You can use the **-Xquickstart** command-line option to improve the performance of short-running applications, especially for applications in which processing is not concentrated into a 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. **-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 compiler” on page 318 for more information.

JVM behavior during idle periods

From Service Refresh 5, 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.

Chapter 27. The Diagnostics Collector

The Diagnostics Collector gathers the Java diagnostics files for a problem event.

Introduction to the Diagnostics Collector

The Diagnostics Collector gathers the Java diagnostics files for a problem event.

The Java runtime produces multiple diagnostics files in response to events such as General Protection Faults, out of memory conditions or receiving unexpected operating system signals. The Diagnostics Collector runs just after the Java runtime produces diagnostics files. It searches for system dumps, Java dumps, heap dumps, Java trace dumps and the verbose GC log that match the time stamp for the problem event. If a system dump is found, then optionally the Diagnostics Collector can execute `jextract` to post-process the dump and capture extra information required to analyze system dumps. The Diagnostics Collector then produces a single `.zip` file containing all the diagnostics for the problem event. Steps in the collection of diagnostics are logged in a text file. At the end of the collection process, the log file is copied into the output `.zip` file.

The Diagnostics Collector also has a feature to give warnings if there are JVM settings in place that could prevent the JVM from producing diagnostics. These warnings are produced at JVM start, so that the JVM can be restarted with fixed settings if necessary. The warnings are printed on `stderr` and in the Diagnostics Collector log file. Fix the settings identified by any warning messages before restarting your Java application. Fixing warnings makes it more likely that the right data is available for IBM Support to diagnose a Java problem.

Using the Diagnostics Collector

The Diagnostics Collector is enabled by a command-line option.

The Diagnostics Collector is off by default and is enabled by a JVM command-line option:

```
-Xdiagnosticscollector[:settings=<filename>]
```

Specifying a Diagnostics Collector settings file is optional. By default, the settings file `jre/lib/dc.properties` is used. See “Diagnostics Collector settings” on page 325 for details of the settings available.

If you run a Java program from the command line with the Diagnostics Collector enabled, it produces some console output. The Diagnostics Collector runs asynchronously, in a separate process to the one that runs your Java program. The effect is that output appears after the command-line prompt returns from running your program. If this happens, it does not mean that the Diagnostics Collector has hung. Press `enter` to get the command-line prompt back.

Collecting diagnostics from Java runtime problems

The Diagnostics Collector produces an output file for each problem event that occurs in your Java application.

When you add the command-line option **-Xdiagnosticscollector**, the Diagnostics Collector runs and produces several output .zip files. One file is produced at startup. Another file is produced for each dump event that occurs during the lifetime of the JVM. For each problem event that occurs in your Java application, one .zip file is created to hold all the diagnostics for that event. For example, an application might have multiple `OutOfMemoryErrors` but keep on running. Diagnostics Collector produces multiple .zip files, each holding the diagnostics from one `OutOfMemoryError`.

The output .zip file is written to the current working directory by default. You can specify a different location by setting the `output.dir` property in the settings file, as described in “Diagnostics Collector settings” on page 325. An output .zip file name takes the form:

```
java.<event>.<YYYYMMDD.hhmmss.pid>.zip
```

In this file name, *<event>* is one of the following names:

- abortsignal
- check
- dumpevent
- gpf
- outofmemoryerror
- usersignal
- vmstart
- vmstop

These event names refer to the event that triggered Diagnostics Collector. The name provides a hint about the type of problem that occurred. The default name is *dumpevent*, and is used when a more specific name cannot be given for any reason.

<YYYYMMDD.hhmmss.pid> is a combination of the time stamp of the dump event, and the process ID for the original Java application. *pid* is not the process ID for the Diagnostics Collector.

The Diagnostics Collector copies files that it writes to the output .zip file. It does not delete the original diagnostics information.

When the Diagnostics Collector finds a system dump for the problem event, then by default it runs `jextract` to post-process the dump and gather context information. This information enables later debugging. Diagnostics Collector automates a manual step that is requested by IBM support on most platforms. You can prevent Diagnostics Collector from running `jextract` by setting the property **run.jextract** to **false** in the settings file. For more information, see “Diagnostics Collector settings” on page 325.

The Diagnostics Collector logs its actions and messages in a file named `JavaDiagnosticsCollector.<number>.log`. The log file is written to the current working directory. The log file is also stored in the output .zip file. The *<number>* component in the log file name is not significant; it is added to keep the log file names unique.

The Diagnostics Collector is a Java VM dump agent. It is run by the Java VM in response to the dump events that produce diagnostic files by default. It runs in a

new Java process, using the same version of Java as the VM producing dumps. This ensures that the tool runs the correct version of jextract for any system dumps produced by the original Java process.

Verifying your Java diagnostics configuration

When you enable the command-line option `-Xdiagnosticscollector`, a diagnostics configuration check runs at Java VM start. If any settings disable key Java diagnostics, a warning is reported.

The aim of the diagnostics configuration check is to avoid the situation where a problem occurs after a long time, but diagnostics are missing because they were inadvertently switched off. Diagnostic configuration check warnings are reported on `stderr` and in the Diagnostics Collector log file. A copy of the log file is stored in the `java.check.<timestamp>.<pid>.zip` output file.

If you do not see any warning messages, it means that the Diagnostics Collector has not found any settings that disable diagnostics. The Diagnostics Collector log file stored in `java.check.<timestamp>.<pid>.zip` gives the full record of settings that have been checked.

For extra thorough checking, the Diagnostics Collector can trigger a Java dump. The dump provides information about the command-line options and current Java system properties. It is worth running this check occasionally, as there are command-line options and Java system properties that can disable significant parts of the Java diagnostics. To enable the use of a Java dump for diagnostics configuration checking, set the `config.check.javacore` option to `true` in the settings file. For more information, see “Diagnostics Collector settings.”

For all platforms, the diagnostics configuration check examines environment variables that can disable Java diagnostics. For reference purposes, the full list of current environment variables and their values is stored in the Diagnostics Collector log file.

Checks for operating system settings are carried out on Linux and AIX. On Linux, the core and file size ulimits are checked. On AIX, the settings `fullcore=true` and `pre430core=false` are checked, as well as the core and file size ulimits.

Configuring the Diagnostics Collector

The Diagnostics Collector supports various options that can be set in a properties file.

Diagnostics Collector can be configured by using options that are set in a properties file. By default, the properties file is `jre/lib/dc.properties`. If you do not have access to edit this file, or if you are working on a shared system, you can specify an alternative filename using:

```
-Xdiagnosticscollector:settings=<filename>
```

Using a settings file is optional. By default, Diagnostics Collector gathers all the main types of Java diagnostics files.

Diagnostics Collector settings

The Diagnostics Collector has several settings that affect the way the collector works.

The settings file uses the standard Java properties format. It is a text file with one **property=value** pair on each line. Each supported property controls the Diagnostics Collector in some way. Lines that start with '#' are comments.

Parameters

file.<any_string>=<pathname>

Any property with a name starting **file.** specifies the path to a diagnostics file to collect. You can add any string as a suffix to the property name, as a reminder of which file the property refers to. You can use any number of **file.** properties, so you can tell the Diagnostics Collector to collect a list of custom diagnostic files for your environment. Using **file.** properties does not alter or prevent the collection of all the standard diagnostic files. Collection of standard diagnostic files always takes place.

Custom debugging scripts or software can be used to produce extra output files to help diagnose a problem. In this situation, the settings file is used to identify the extra debug output files for the Diagnostics Collector. The Diagnostics Collector collects the extra debug files at the point when a problem occurs. Using the Diagnostics Collector in this way means that debug files are collected immediately after the problem event, increasing the chance of capturing relevant context information.

output.dir=<output_directory_path>

The Diagnostics Collector tries to write its output .zip file to the output directory path that you specify. The path can be absolute or relative to the working directory of the Java process. If the directory does not exist, the Diagnostics Collector tries to create it. If the directory cannot be created, or the directory is not writeable, the Diagnostics Collector defaults to writing its output .zip file to the current working directory.

Note: On Windows systems, Java properties files use backslash as an escape character. To specify a backslash as part of Windows path name, use a double backslash '\\' in the properties file.

loglevel.file=<level>

This setting controls the amount of information written to the Diagnostics Collector log file. The default setting for this property is **config**. Valid levels are:

off No information reported.

severe Errors are reported.

warning

Report warnings in addition to information reported by **severe**.

info More detailed information in addition to that reported by **warning**.

config Configuration information reported in addition to that reported by **info**. This is the default reporting level.

fine Tracing information reported in addition to that reported by **config**.

finer Detailed tracing information reported in addition to that reported by **fine**.

finest Report even more tracing information in addition to that reported by **finer**.

all Report everything.

loglevel.console=<level>

Controls the amount of information written by the Diagnostics Collector to stderr. Valid values for this property are as described for loglevel.file. The default setting for this property is **warning**.

settings.id=<identifier>

Allows you to set an identifier for the settings file. If you set loglevel.file to **fine** or **lower**, the **settings.id** is recorded in the Diagnostics Collector log file as a way to check that your settings file is loaded as expected.

config.check.javacore={true | false}

Set **config.check.javacore=true** to enable a Java dump for the diagnostics configuration check at virtual machine start-up. The check means that the virtual machine start-up takes more time, but it enables the most thorough level of diagnostics configuration checking.

run.jextract=false

Set this option to prevent the Diagnostics Collector running jextract on detected System dumps.

Known limitations

There are some known limitations for the Diagnostics Collector.

If Java programs do not start at all on your system, for example because of a Java runtime installation problem or similar issue, the Diagnostics Collector cannot run.

The Diagnostics Collector does not respond to additional **-Xdump** settings that specify extra dump events requiring diagnostic information. For example, if you use **-Xdump** to produce dumps in response to a particular exception being thrown, the Diagnostics Collector does not collect the dumps from this event.

Chapter 28. Garbage Collector diagnostics

This section describes how to diagnose garbage collection.

The topics that are discussed in this chapter are:

- “How do the garbage collectors work?”
- “Common causes of perceived leaks”
- “-verbose:gc logging” on page 330
- “-Xtgc tracing” on page 340

How do the garbage collectors work?

Garbage collection identifies and frees previously allocated storage that is no longer in use. An understanding of the way that the Garbage Collector works will help you to diagnose problems.

Read Chapter 2, “Memory management,” on page 7 to get a full understanding of the Garbage Collector. A short introduction to the Garbage Collector is given here.

The JVM includes a Memory Manager, which manages the Java heap. The Memory Manager allocates space from the heap as objects are instantiated, keeping a record of where the remaining free space in the heap is located. When free space in the heap is low and an object allocation cannot be satisfied, an allocation failure is triggered and a garbage collection cycle is started. When this process is complete, the memory manager tries the allocation that it could not previously satisfy again.

An application can request a manual garbage collection at any time, but this action is not recommended. See “How to coexist with the Garbage Collector” on page 23.

Common causes of perceived leaks

When a garbage collection cycle starts, the Garbage Collector must locate all objects in the heap that are still in use or “live”. When this has been done, any objects that are not in the list of live objects are unreachable. They are garbage, and can be collected.

The key here is the condition *unreachable*. The Garbage Collector traces all references that an object makes to other objects. Any such reference automatically means that an object is *reachable* and not garbage. Therefore, if the objects of an application make reference to other objects, those other objects are live and cannot be collected. However, obscure references sometimes exist that the application overlooks. These references are reported as memory leaks.

Listeners

By installing a listener, you are effectively attaching your object to a static reference that is in the listener.

Your object cannot be collected while the listener is available. When you have finished using the object, you must uninstall the listener which your object is attached to.

Hash tables

Anything that is added to a hash table, either directly or indirectly, from an instance of your object, creates a reference to your object from the hashed object. Hashed objects cannot be collected unless they are explicitly removed from any hash table to which they have been added.

Hash tables are common causes of perceived leaks. If an object is placed into a hash table, that object and all the objects that it references are reachable.

Static class data

Static class data exists independently of instances of your object. Anything that it points to cannot be collected even if no instances of your class are present that contain the static data.

JNI references

Objects that are passed from the JVM to native code using the JNI interface must have a reference.

When using JNI, a reference to objects passed from the JVM to native code must be held in the JNI code of the JVM. Without this reference, the Garbage Collector cannot trace live objects referenced from native code. The object references must be cleared explicitly by the native code application before they can be collected.

See the JNI documentation at (<http://java.sun.com>) for more information.

Objects with finalizers

Objects that have finalizers cannot be collected until the finalizer has run.

Finalizers run on a separate thread, therefore their execution might be delayed, or not occur at all. This behavior can give the impression that your unused object is not being collected. You might also believe that a memory leak has occurred.

The IBM Garbage Collector (GC) does *not* collect garbage unless it must. The GC does not necessarily collect all garbage when it runs. The GC might not collect garbage if you manually start it, using `System.gc()`. The GC is designed to run infrequently and quickly, because application threads are stopped while the garbage is collected.

See “How to coexist with the Garbage Collector” on page 23 for more details.

-verbose:gc logging

Verbose logging is intended as the first tool to be used when attempting to diagnose garbage collector problems; more detailed analysis can be performed by calling one or more `-Xtgc` (trace garbage collector) traces.

Note that 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 Chapter 2, “Memory management,” on page 7 if necessary.

By default, **-verbose:gc** output is written to `stderr`. You can redirect the output to a file using the **-Xverbosegclog** command-line option (see “Garbage Collector command-line options” on page 453 for more information).

Global collections

An example of the output produced when a global collection is triggered.

The example is:

```
<gc type="global" id="5" totalid="5" intervalms="18.880">
  <compaction movecount="9282" movebytes="508064" reason="forced compaction" />

  <expansion type="tenured" amount="1048576" newsize="3145728" timetaken="0.011"
    reason="insufficient free space following gc" />
    <refs_cleared soft="0" weak="0" phantom="0" />
    <finalization objectsqueued="0" />
    <timesms mark="7.544" sweep="0.088" compact="9.992" total="17.737" />
    <tenured freebytes="1567256" totalbytes="3145728" percent="49" >
      <soa freebytes="1441816" totalbytes="3020288" percent="47" />
      <loa freebytes="125440" totalbytes="125440" percent="100" />
    </tenured>
  </gc>
```

<gc> Indicates that a garbage collection was triggered on the heap. **Type="global"** indicates that the collection was global (mark, sweep, possibly compact). The **id** attribute gives the occurrence number of this global collection. The **totalid** indicates the total number of garbage collections (of all types) that have taken place. Currently this number is the sum of the number of global collections and the number of scavenger collections. **intervalms** gives the number of milliseconds since the previous global collection.

<compaction>

Shows the number of objects that were moved during compaction and the total number of bytes these objects represented. The reason for the compaction is also shown. In this case, the compaction was forced, because **-Xcompactgc** was specified on the command line. This line is displayed only if compaction occurred during the collection.

<expansion>

Indicates that during the handling of the allocation (but after the garbage collection), a heap expansion was triggered. The area expanded, the amount by which the area was increased (in bytes), its new size, the time taken to expand, and the reason for the expansion are shown.

<refs_cleared>

Provides information relating to the number of Java Reference objects that were cleared during the collection. In this example, no references were cleared.

<finalization>

Provides information detailing the number of objects containing finalizers that were enqueued for VM finalization during the collection.

Note: The number of objects is not equal to the number of finalizers that were run during the collection, because finalizers are scheduled by the VM.

<timesms>

Provides information detailing the times taken for each of the mark, the sweep, and then compact phases, as well as the total time taken. When compaction was not triggered, the number returned for compact is zero.

<tenured>

Indicates the status of the tenured area following the collection. If running in generational mode, a **<nursery>** line is output, showing the status of the active new area.

Garbage collection triggered by System.gc()

Java programs can trigger garbage collections to occur manually by calling the method `System.gc()`.

-verbose:gc output produced by `System.gc()` calls is similar to:

```
<sys id="1" timestamp="Jul 15 12:56:26 2005" intervals="0.000">
  <time exclusiveaccessms="0.018" />

  <tenured freebytes="821120" totalbytes="4194304" percent="19" >
    <soa freebytes="611712" totalbytes="3984896" percent="15" />
    <loa freebytes="209408" totalbytes="209408" percent="100" />
  </tenured>
  <gc type="global" id="1" totalid="1" intervals="0.000">
    <classloadersunloaded count="0" timetakenms="0.012" />
    <refs_cleared soft="0" weak="4" phantom="0" />
    <finalization objectsqueued="6" />
    <timesms mark="3.065" sweep="0.138" compact="0.000" total="3.287" />
    <tenured freebytes="3579072" totalbytes="4194304" percent="85" >
      <soa freebytes="3369664" totalbytes="3984896" percent="84" />
      <loa freebytes="209408" totalbytes="209408" percent="100" />
    </tenured>
  </gc>
  <tenured freebytes="3579072" totalbytes="4194304" percent="85" >
    <soa freebytes="3369664" totalbytes="3984896" percent="84" />
    <loa freebytes="209408" totalbytes="209408" percent="100" />
  </tenured>

  <time totalms="3.315" />
</sys>
```

<gc type="global">

Indicates that, as a result of the `System.gc()` call, a global garbage collection was triggered. The contents of the **<gc>** tag for a global collection are explained in detail in “Global collections” on page 331 with the exception of the **<classloadersunloaded>** tag, which indicates how many unused class loaders were collected.

<sys> Indicates that a `System.gc()` has occurred. The **id** attribute gives the number of this `System.gc()` call; in this case, this is the first such call in the life of this VM. **timestamp** gives the local timestamp when the `System.gc()` call was made and **intervals** gives the number of milliseconds that have elapsed since the previous `System.gc()` call. In this case, because this is the first such call, the number returned is zero.

<tenured>

Shows the occupancy levels of the different heap areas before the garbage collection - both the small object area (SOA) and the large object area (LOA).

<time exclusiveaccessms=>

Shows the amount of time taken to obtain exclusive VM access. A further optional line `<warning details="exclusive access time includes previous garbage collections" />` might occasionally be displayed, to inform you that the following garbage collection was queued because the allocation failure was triggered while another thread was already performing a garbage collection. Typically, this first collection will have freed enough heap space to satisfy both allocation requests (the original one that triggered the garbage collection and the subsequently queued allocation request). However, sometimes this is not the case and another garbage collection is triggered almost immediately. This additional line

informs you that the pause time displayed might be slightly misleading unless you are aware of the underlying threading used.

<time>

Shows the total amount of time taken to handle the System.gc() call (in milliseconds).

Allocation failures

When an attempt is made to allocate to the heap but insufficient memory is available, an allocation failure is triggered. The output produced depends on the area of the heap in which the allocation failure occurred.

New area allocation failures

This example shows you the information produced when an allocation failure occurs in the new area (nursery).

```
<af type="nursery" id="28" timestamp="Jul 15 13:11:45 2005" intervals="65.016">
  <minimum requested_bytes="520" />
  <time exclusiveaccessms="0.018" />

  <nursery freebytes="0" totalbytes="8239104" percent="0" />
  <tenured freebytes="5965800" totalbytes="21635584" percent="27" >
    <soa freebytes="4884456" totalbytes="20554240" percent="23" />
    <loa freebytes="1081344" totalbytes="1081344" percent="100" />
  </tenured>
  <gc type="scavenger" id="28" totalid="30" intervals="65.079">
    <expansion type="nursery" amount="1544192" newsize="9085952" timetaken="0.017"
      reason="excessive time being spent scavenging" />
    <flipped objectcount="16980" bytes="2754828" />
    <tenured objectcount="12996" bytes="2107448" />
    <refs_cleared soft="0" weak="0" phantom="0" />
    <finalization objectsqueued="0" />
    <scavenger tilratio="70" />
    <nursery freebytes="6194568" totalbytes="9085952" percent="68" tenureage="1" />
    <tenured freebytes="3732376" totalbytes="21635584" percent="17" >
      <soa freebytes="2651032" totalbytes="20554240" percent="12" />
      <loa freebytes="1081344" totalbytes="1081344" percent="100" />
    </tenured>
    <time totalms="27.043" />
  </gc>
  <nursery freebytes="6194048" totalbytes="9085952" percent="68" />
  <tenured freebytes="3732376" totalbytes="21635584" percent="17" >
    <soa freebytes="2651032" totalbytes="20554240" percent="12" />
    <loa freebytes="1081344" totalbytes="1081344" percent="100" />
  </tenured>

  <time totalms="27.124" />
</af>
```

<af type="nursery">

Indicates that an allocation failure has occurred when attempting to allocate to the new area. The **id** attribute shows the index of the type of allocation failure that has occurred. **timestamp** shows a local timestamp at the time of the allocation failure. **intervals** shows the number of milliseconds elapsed since the previous allocation failure of that type.

<minimum>

Shows the number of bytes requested by the allocation that triggered the failure. Following the garbage collection, **freebytes** might drop by more than this amount. The reason is that the free list might have been discarded or the Thread Local Heap (TLH) refreshed.

<gc>

Indicates that, as a result of the allocation failure, a garbage collection was

triggered. In this example, a scavenger collection occurred. The contents of the <gc> tag are explained in detail in “Scavenger collections.”

<nursery> and <tenured>

The first set of <nursery> and <tenured> tags show the status of the heaps at the time of the allocation failure that triggered garbage collection. The second set of tags shows the status of the heaps after the garbage collection has occurred. The third set of tags shows the status of the different heap areas following the successful allocation.

<time> Shows the total time taken to handle the allocation failure.

Tenured allocation failures

This example shows you the output produced when an allocation occurs in the tenured area.

```
<af type="tenured" id="2" timestamp="Jul 15 13:17:11 2005" intervals="450.057">
  <minimum requested_bytes="32" />
  <time exclusiveaccessms="0.015" />

  <tenured freebytes="104448" totalbytes="2097152" percent="4" >
    <soa freebytes="0" totalbytes="1992704" percent="0" />
    <loa freebytes="104448" totalbytes="104448" percent="100" />
  </tenured>
  <gc type="global" id="4" totalid="4" intervals="217.002">
    <expansion type="tenured" amount="1048576" newsize="3145728" timetaken="0.008"
      reason="insufficient free space following gc" />
    <refs_cleared soft="0" weak="0" phantom="0" />
    <finalization objectsqueued="5" />
    <timesms mark="4.960" sweep="0.113" compact="0.000" total="5.145" />
    <tenured freebytes="1612176" totalbytes="3145728" percent="51" >
      <soa freebytes="1454992" totalbytes="2988544" percent="48" />
      <loa freebytes="157184" totalbytes="157184" percent="100" />
    </tenured>
  </gc>
  <tenured freebytes="1611632" totalbytes="3145728" percent="51" >
    <soa freebytes="1454448" totalbytes="2988544" percent="48" />
    <loa freebytes="157184" totalbytes="157184" percent="100" />
  </tenured>

  <time totalms="5.205" />
</af>
```

Scavenger collections

This example shows you the output produced when a scavenger collection is triggered.

To understand when the Garbage Collector starts a scavenger collection, see “Generational Concurrent Garbage Collector” on page 19

```
<gc type="scavenger" id="11" totalid="11" intervals="46.402">
  <failed type="tenured" objectcount="24" bytes="43268" />
  <flipped objectcount="523" bytes="27544" />
  <tenured objectcount="0" bytes="0" />
  <refs_cleared soft="0" weak="0" phantom="0" />
  <finalization objectsqueued="0" />
  <scavenger tiltratio="67" />
  <nursery freebytes="222208" totalbytes="353280" percent="62" tenureage="2" />
  <tenured freebytes="941232" totalbytes="1572864" percent="59" >
    <soa freebytes="862896" totalbytes="1494528" percent="57" />
    <loa freebytes="78336" totalbytes="78336" percent="100" />
  </tenured>
  <time totalms="0.337" />
</gc>
```

- <gc>** Indicates that a garbage collection has been triggered. The `type="scavenger"` attribute indicates that the collection is a scavenger collection. The `id` attribute shows how many of this type of collection have taken place. The `totalid` attribute shows the total number of garbage collections of all types that have taken place, including this one. `intervalms` gives the amount of time in milliseconds since the last collection of this type.
- <failed type="tenured">**
Indicates that the scavenger failed to move some objects into the old or "tenured" area during the collection. The output shows the number of objects that were not moved, and the total bytes represented by these objects. If `<failed type="flipped">` is shown, the scavenger failed to move or "flip" certain objects into the survivor space.
- <flipped>**
Shows the number of objects that were flipped into the survivor space during the scavenger collection, together with the total number of bytes flipped.
- <scavenger tiltratio="x" />**
Shows the percentage of the tilt ratio following the last scavenge event and space adjustment. The scavenger redistributes memory between the allocate and survivor areas 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. For further information about the tilt ratio, see "Tilt ratio" on page 20.
- <tenured>**
Shows the number of objects that were moved into the tenured area during the scavenger collection, together with the total number of bytes tenured.
- <nursery>**
Shows the amount of free and total space in the nursery area after a scavenge event. The output also shows the number of times an object must be flipped in order to be tenured. This number is the tenure age, and is adjusted dynamically.
- <time>** Shows the total time taken to perform the scavenger collection, in milliseconds.

In certain situations, a number of additional lines can be generated during a scavenger collection:

- If a scavenger collection fails, an additional `<warning details="aborted collection" />` line is included. Failure might occur if the new area was excessively tilted with a full tenured area, and certain objects were not copied or tenured.
- If it is not possible to tenure an object, an expansion of the tenured area might be triggered. This event is shown as a separate line of `-verbose:gc`.
- If "remembered set overflow" or "scan cache overflow" occurred during a scavenger collection, these events are shown as separate lines of `-verbose:gc`.
- If all of the new space is resized following a scavenger collection, additional lines are added to `-verbose:gc`.

Concurrent garbage collection

When running with concurrent garbage collection, several additional `-verbose:gc` outputs are displayed.

Concurrent sweep completed

This output shows that the concurrent sweep process (started after the previous garbage collection completed) has finished. The amount of bytes swept and the amount of time taken is shown.

```
<con event="completed sweep" timestamp="Jul 15 13:52:08 2005">
  <stats bytes="0" time="0.004" />
</con>
```

Concurrent kickoff

This example shows you the output produced when the concurrent mark process is triggered.

```
<con event="kickoff" timestamp="Nov 25 10:18:52 2005">
  <stats tenurfreebytes="2678888" tracetarget="21107394"
    kickoff="2685575" tracerate="8.12" />
</con>
```

This output shows that concurrent mark was kicked off, and gives a local timestamp for this. Statistics are produced showing the amount of free space in the tenured area, the target amount of tracing to be performed by concurrent mark, the kickoff threshold at which concurrent is triggered, and the initial trace rate. The trace rate represents the amount of tracing each mutator thread should perform relative to the amount of space it is attempting to allocate in the heap. In this example, a mutator thread that allocates 20 bytes will be required to trace $20 * 8.12 = 162$ bytes. If also running in generational mode, an additional **nurseryfreebytes**= attribute is displayed, showing the status of the new area as concurrent mark was triggered.

Allocation failures during concurrent mark

When an allocation failure occurs during concurrent mark, tracing is disrupted. If the allocation is "aborted", the trace data is discarded. If the allocation is "halted", tracing resumes during a subsequent collection.

Concurrent aborted:

This example shows the output produced when concurrent mark is aborted.

```
<af type="tenured" id="4" timestamp="Jul 15 14:08:28 2005" intervals="17.479">
  <minimum requested_bytes="40" />
  <time exclusiveaccessms="0.041" />

  <tenured freebytes="227328" totalbytes="5692928" percent="3" >
    <soa freebytes="0" totalbytes="5465600" percent="0" />
    <loa freebytes="227328" totalbytes="227328" percent="100" />
  </tenured>
  <con event="aborted" />
</gc type="global" id="6" totalid="6" intervals="17.541">
  <warning details="completed sweep to facilitate expansion" />
  <expansion type="tenured" amount="2115584" newsize="7808512" timetaken="0.010"
    reason="insufficient free space following gc" />
  <refs_cleared soft="0" weak="0" phantom="0" />
  <finalization objectsqueued="0" />
  <timesms mark="17.854" sweep="0.201" compact="0.000" total="18.151" />
  <tenured freebytes="2342952" totalbytes="7808512" percent="30" >
    <soa freebytes="2108968" totalbytes="7574528" percent="27" />
    <loa freebytes="233984" totalbytes="233984" percent="100" />
  </tenured>
</gc>
<tenured freebytes="2340904" totalbytes="7808512" percent="29" >
  <soa freebytes="2106920" totalbytes="7574528" percent="27" />
  <loa freebytes="233984" totalbytes="233984" percent="100" />
```



```

</tenured>

<time totalms="18.252" />
</af>

<con event="aborted">
    Shows that, as a result of the allocation failure, concurrent mark tracing
    was aborted.

```

Concurrent halted:

This example shows the output produced when concurrent mark is halted.

```

<af type="tenured" id="5" timestamp="Jul 15 14:08:28 2005" intervalms="249.9
55">
    <minimum requested_bytes="32" />
    <time exclusiveaccessms="0.022" />

    <tenured freebytes="233984" totalbytes="7808512" percent="2" >
        <soa freebytes="0" totalbytes="7574528" percent="0" />
        <loa freebytes="233984" totalbytes="233984" percent="100" />
    </tenured>
    <con event="halted" mode="trace only">
        <stats tracetarget="2762287">
            <traced total="137259" mutators="137259" helpers="0" percent="4" />
            <cards cleaned="0" kickoff="115809" />
        </stats>
    </con>
    <con event="final card cleaning">
        <stats cardscleaned="16" traced="2166272" durationms="22.601" />
    </con>
    <gc type="global" id="7" totalid="7" intervalms="272.635">
        <warning details="completed sweep to facilitate expansion" />
        <expansion type="tenured" amount="3013120" newsize="10821632" timetaken="0.015"
            reason="insufficient free space following gc" />
        <refs_cleared soft="0" weak="0" phantom="0" />
        <finalization objectsqueued="0" />
        <timesms mark="2.727" sweep="0.251" compact="0.000" total="3.099" />
        <tenured freebytes="3247120" totalbytes="10821632" percent="30" >
            <soa freebytes="3031056" totalbytes="10605568" percent="28" />
            <loa freebytes="216064" totalbytes="216064" percent="100" />
        </tenured>
    </gc>
    <tenured freebytes="3245072" totalbytes="10821632" percent="29" >
        <soa freebytes="3029008" totalbytes="10605568" percent="28" />
        <loa freebytes="216064" totalbytes="216064" percent="100" />
    </tenured>

    <time totalms="25.800" />
</af>

```

```

<con event="halted">
    Shows that concurrent mark tracing was halted as a result of the allocation
    failure. The tracing target is shown, together with the amount that was
    performed, both by mutator threads and the concurrent mark background
    thread. The percentage of the trace target traced is shown. The number of
    cards cleaned during concurrent marking is also shown, with the
    free-space trigger level for card cleaning. Card cleaning occurs during
    concurrent mark after all available tracing has been exhausted.

```

```

<con event="final card cleaning">
    Indicates that final card cleaning occurred before the garbage collection
    was triggered. The number of cards cleaned during the process and the
    number of bytes traced is shown, along with the total time taken by the
    process.

```

Concurrent collection:

If concurrent mark completes all tracing and card cleaning, a concurrent collection is triggered.

The output produced by this concurrent collection is shown:

```
<con event="collection" id="15" timestamp="Jul 15 15:13:18 2005"
      intervalms="1875.113">
  <time exclusiveaccessms="2.080" />

  <tenured freebytes="999384" totalbytes="137284096" percent="0" >
    <soa freebytes="999384" totalbytes="137284096" percent="0" />
    <loa freebytes="0" totalbytes="0" percent="0" />
  </tenured>
  <stats tracetarget="26016936">
    <traced total="21313377" mutators="21313377" helpers="0" percent="81" />
    <cards cleaned="14519" kickoff="1096607" />
  </stats>
  <con event="completed full sweep" timestamp="Jul 15 15:13:18 2005">
    <stats sweepbytes="0" sweeptime="0.009" connectbytes="5826560"
      connecttime="0.122" />
  </con>
  <con event="final card cleaning">
    <stats cardscleaned="682" traced="302532" durationms="3.053" />
  </con>
  <gc type="global" id="25" totalid="25" intervalms="1878.375">
    <expansion type="tenured" amount="19365376" newsize="156649472"
      timetaken="0.033" reason="insufficient free space following gc" />
    <refs_cleared soft="0" weak="0" phantom="0" />
    <finalization objectsqueued="0" />
    <timesms mark="49.014" sweep="0.143" compact="0.000" total="50.328" />
    <tenured freebytes="46995224" totalbytes="156649472" percent="30" >
      <soa freebytes="46995224" totalbytes="156649472" percent="30" />
      <loa freebytes="0" totalbytes="0" percent="0" />
    </tenured>
  </gc>
  <tenured freebytes="46995224" totalbytes="156649472" percent="30" >
    <soa freebytes="46995224" totalbytes="156649472" percent="30" />
    <loa freebytes="0" totalbytes="0" percent="0" />
  </tenured>

  <time totalms="55.844" />
</con>
```

<con event="collection">

Shows that a concurrent collection has been triggered. The **id** attribute shows the number of this concurrent collection, next is a local timestamp, and the number of milliseconds since the previous concurrent collection is displayed.

<stats>

Shows the tracing statistics for the concurrent tracing that has taken place previously. The target amount of tracing is shown, together with the amount that took place (both by mutators threads and helper threads). Information is displayed showing the number of cards in the card table that were cleaned during the concurrent mark process, and the heap occupancy level at which card cleaning began.

<con event="completed full sweep">

Shows that the full concurrent sweep of the heap was completed. The number of bytes of the heap swept is displayed with the amount of time taken, the amount of bytes swept that were connected together, and the time taken to do this.

<con event="final card cleaning">

Shows that final card cleaning has been triggered. The number of cards cleaned is displayed, together with the number of milliseconds taken to do so.

Following these statistics, a normal global collection is triggered.

System.gc() calls during concurrent mark

This example shows the output produced when a System.gc() call is made during concurrent mark.

```
<sys id="6" timestamp="Jul 15 15:57:49 2005" intervalms="179481.748">
  <time exclusiveaccessms="0.030" />
  <tenured freebytes="1213880" totalbytes="152780800" percent="0" >
    <soa freebytes="1213880" totalbytes="152780800" percent="0" />
    <loa freebytes="0" totalbytes="0" percent="0" />
  </tenured>
  <con event="completed full sweep" timestamp="Jul 15 15:57:49 2005">
    <stats sweepbytes="0" sweeptime="0.009" connectbytes="3620864"
      connecttime="0.019" />
  </con>
  <con event="halted" mode="clean trace">
    <stats tracetarget="31394904">
      <traced total="23547612" mutators="23547612" helpers="0" percent="75" />
      <cards cleaned="750" kickoff="1322108" />
    </stats>
  </con>
  <con event="final card cleaning">
    <stats cardscleaned="10588" traced="5202828" durationms="48.574" />
  </con>
  <gc type="global" id="229" totalid="229" intervalms="1566.763">
    <warning details="completed sweep to facilitate compaction" />
    <compaction movecount="852832" movebytes="99934168" reason="compact on
      aggressive collection" />
    <classloadersunloaded count="0" timetakenms="0.009" />
    <refs cleared soft="0" weak="0" phantom="0" />
    <finalization objectsqueued="0" />
    <timesms mark="44.710" sweep="13.046" compact="803.052" total="863.470" />
    <tenured freebytes="52224264" totalbytes="152780800" percent="34" >
      <soa freebytes="52224264" totalbytes="152780800" percent="34" />
      <loa freebytes="0" totalbytes="0" percent="0" />
    </tenured>
  </gc>
  <tenured freebytes="52224264" totalbytes="152780800" percent="34" >
    <soa freebytes="52224264" totalbytes="152780800" percent="34" />
    <loa freebytes="0" totalbytes="0" percent="0" />
  </tenured>
  <time totalms="863.542" />
</sys>
```

This output shows that a System.gc() call was made after concurrent mark had started. In this case, enough tracing had been performed for the work to be reused, so that concurrent mark is halted rather than aborted. The results for final card-cleaning are also shown.

Timing problems during garbage collection

If the clock on your workstation is experiencing problems, time durations in verbosegc might be incorrectly output as 0.000 ms.

This example shows the output produced if the clock is experiencing problems.

```
<af type="nursery" id="89" timestamp="Dec 11 19:10:54 2006" intervalms="285.778">
  <minimum requested_bytes="24" />
  <time exclusiveaccessms="872.224" />
```

```

<warning details="exclusive access time includes previous garbage collections" />
<nursery freebytes="0" totalbytes="46418944" percent="0" />
<tenured freebytes="310528840" totalbytes="1505755136" percent="20" >
  <soa freebytes="282498888" totalbytes="1477725184" percent="19" />
  <loa freebytes="28029952" totalbytes="28029952" percent="100" />
</tenured>
<gc type="scavenger" id="89" totalid="92" intervalms="287.023">
  <flipped objectcount="230263" bytes="14110324" />
  <tenured objectcount="175945" bytes="10821424" />
  <refs_cleared soft="0" weak="0" phantom="0" />
  <finalization objectsqueued="0" />
  <scavenger tiltratio="72" />
  <nursery freebytes="32717416" totalbytes="48452096" percent="67"
    tenureage="1" />
  <tenured freebytes="298206144" totalbytes="1505755136" percent="19" >
    <soa freebytes="270176192" totalbytes="1477725184" percent="18" />
    <loa freebytes="28029952" totalbytes="28029952" percent="100" />
  </tenured>
  <time totalms="147.061" />
</gc>
<nursery freebytes="32715368" totalbytes="48452096" percent="67" />
<tenured freebytes="298206144" totalbytes="1505755136" percent="19" >
  <soa freebytes="270176192" totalbytes="1477725184" percent="18" />
  <loa freebytes="28029952" totalbytes="28029952" percent="100" />
</tenured>
<warning details="clock error detected in time totalms" />
<time totalms="0.000" />
</af>

```

The warning message clock error detected in time totalms indicates that when verbosegc sampled the system time at the end of the garbage collection, the value returned was earlier than the start time. This time sequence is clearly wrong, and a warning message is output. Possible causes for this error include the following:

- Your system is synchronizing with an external NTP server.
- Workstations in a middleware cluster are synchronizing their clocks with each other.

To work around this problem, disable the updating of your system time while the Java program is running.

-Xtgc tracing

By enabling one or more TGC (trace garbage collector) traces, more detailed garbage collection information than that displayed by **-verbose:gc** will be shown.

This section summarizes the different **-Xtgc** traces available. The output is written to stdout. More than one trace can be enabled simultaneously by separating the parameters with commas, for example **-Xtgc:backtrace,compaction**.

-Xtgc:backtrace

This trace shows information tracking which thread triggered the garbage collection.

For a System.gc() this might be similar to:

```
"main" (0x0003691C)
```

This shows that the GC was triggered by the thread with the name "main" and osThread 0x0003691C.

One line is printed for each global or scavenger collection, showing the thread that triggered the GC.

-Xtgc:compaction

This trace shows information relating to compaction.

The trace is similar to:

```
Compact(3): reason = 7 (forced compaction)
Compact(3): Thread 0, setup stage: 8 ms.
Compact(3): Thread 0, move stage: handled 42842 objects in 13 ms, bytes moved 2258028.
Compact(3): Thread 0, fixup stage: handled 0 objects in 0 ms, root fixup time 1 ms.
Compact(3): Thread 1, setup stage: 0 ms.
Compact(3): Thread 1, move stage: handled 35011 objects in 8 ms, bytes moved 2178352.
Compact(3): Thread 1, fixup stage: handled 74246 objects in 13 ms, root fixup time 0 ms.
Compact(3): Thread 2, setup stage: 0 ms.
Compact(3): Thread 2, move stage: handled 44795 objects in 32 ms, bytes moved 2324172.
Compact(3): Thread 2, fixup stage: handled 6099 objects in 1 ms, root fixup time 0 ms.
Compact(3): Thread 3, setup stage: 8 ms.
Compact(3): Thread 3, move stage: handled 0 objects in 0 ms, bytes moved 0.
Compact(3): Thread 3, fixup stage: handled 44797 objects in 7 ms, root fixup time 0 ms.
```

This trace shows that compaction occurred during the third global GC, for reason "7". In this case, four threads are performing compaction. The trace shows the work performed by each thread during setup, move, and fixup. The time for each stage is shown together with the number of objects handled by each thread.

-Xtgc:concurrent

This trace displays basic extra information about the concurrent mark helper thread.

```
<CONCURRENT GC BK thread 0x0002645F activated after GC(5)>
```

```
<CONCURRENT GC BK thread 0x0002645F (started after GC(5)) traced 25435>
```

This trace shows when the background thread was activated, and the amount of tracing it performed (in bytes).

-Xtgc:dump

This trace shows extra information following the sweep phase of a global garbage collection.

This is an extremely large trace – a sample of one GC's output is:

```
<GC(4) 13F9FE44 freelen=x000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA0140 freelen=x00000010>
<GC(4) 13FA0150 freelen=x00000050 -- x0000001C java/lang/String>
<GC(4) 13FA0410 freelen=x000002C4 -- x00000024 spec/jbb/infra/Collections/
    longBTreeNode>
<GC(4) 13FA0788 freelen=x00000004 -- x00000050 java/lang/Object[]>
<GC(4) 13FA0864 freelen=x00000010>
<GC(4) 13FA0874 freelen=x0000005C -- x0000001C java/lang/String>
<GC(4) 13FA0B4C freelen=x000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA0E48 freelen=x00000010>
<GC(4) 13FA0E58 freelen=x00000068 -- x0000001C java/lang/String>
<GC(4) 13FA1148 freelen=x000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA1444 freelen=x00000010>
<GC(4) 13FA1454 freelen=x0000006C -- x0000001C java/lang/String>
<GC(4) 13FA174C freelen=x000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA1A48 freelen=x00000010>
<GC(4) 13FA1A58 freelen=x00000054 -- x0000001C java/lang/String>
<GC(4) 13FA1D20 freelen=x000002C4 -- x00000038 spec/jbb/Stock>
```

```

<GC(4) 13FA201C freelen=x00000010>
<GC(4) 13FA202C freelen=x00000044 -- x0000001C java/lang/String>
<GC(4) 13FA22D4 freelen=x0000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA25D0 freelen=x00000010>
<GC(4) 13FA25E0 freelen=x00000048 -- x0000001C java/lang/String>
<GC(4) 13FA2890 freelen=x0000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA2B8C freelen=x00000010>
<GC(4) 13FA2B9C freelen=x00000068 -- x0000001C java/lang/String>
<GC(4) 13FA2E8C freelen=x0000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA3188 freelen=x00000010>

```

A line of output is printed for every free chunk in the system, including dark matter (free chunks that are not on the free list for some reason, usually because they are too small). Each line contains the base address and the size in bytes of the chunk. If the chunk is followed in the heap by an object, the size and class name of the object is also printed.

-Xtgc:excessiveGC

This trace shows statistics for garbage collection cycles.

After a garbage collection cycle has completed, a trace entry is produced:

```

excessiveGC: gcid="10" intimems="122.269" outtimems="1.721" \
             percent="98.61" averagepercent="37.89"

```

This trace shows how much time was spent performing garbage collection and how much time was spent out of garbage collection. In this example, garbage collection cycle 10 took 122.269 ms to complete and 1.721 ms passed between collections 9 and 10. These statistics show that garbage collection accounted for 98.61% of the time from the end of collection 9 to the end of collection 10. The average time spent in garbage collection is 37.89%.

When the average time in garbage collection reaches 95%, extra trace entries are produced:

```

excessiveGC: gcid="65" percentreclaimed="1.70" freedelta="285728" \
             activessize="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 453).

When the average time in garbage collection reaches 95% and the percentage of free space reclaimed by a collection drops below 3%, another trace entry is produced:

```

excessiveGC: gcid="65" percentreclaimed="1.70" minimum="3.00" excessive gc raised

```

The JVM will then throw an OutOfMemoryError.

-Xtgc:freelist

Before a garbage collection, this trace prints information about the free list and allocation statistics since the last GC.

The trace prints the number of items on the free list, including “deferred” entries (with the scavenger, the unused semispace is a deferred free list entry). For TLH and non-TLH allocations, this prints the total number of allocations, the average allocation size, and the total number of bytes discarded during allocation. For

non-TLH allocations, also included is the average number of entries that were searched before a sufficiently large entry was found.

```
*8* free      0
*8* deferred  0
total         0
<Alloc TLH: count 3588, size 3107, discard 31>
< non-TLH: count 6219, search 0, size 183, discard 0>
```

-Xtgc:parallel

This trace shows statistics about the activity of the parallel threads during the mark and sweep phases of a global garbage collection.

```
Mark:  busy  stall  tail  acquire  release
0:      30    30    0        0        3
1:      53     7    0       91       94
2:      29    31    0       37       37
3:      37    24    0      243      237
Sweep: busy  idle sections 127 merge 0
0:      10     0    96
1:       8     1     0
2:       8     1    31
3:       8     1     0
```

This trace shows four threads (0-3), together with the work done by each thread during the mark and sweep phases of garbage collection.

For the mark phase of garbage collection, the time spent in the "busy", "stalled", and "tail" states is shown (in milliseconds). The number of work packets each thread acquired and released during the mark phase is also shown.

For the sweep phase of garbage collection, the time spent in the "busy" and "idle" states is shown (in milliseconds). The number of sweep chunks processed by each thread is also shown, including the total (127). The total merge time is also shown (0ms).

-Xtgc:references

This trace shows activity relating to reference handling during garbage collections.

```
enqueueing ref sun/misc/SoftCache$ValueCell@0x1564b5ac -> 0x1564b4c8
enqueueing ref sun/misc/SoftCache$ValueCell@0x1564b988 -> 0x1564b880
enqueueing ref sun/misc/SoftCache$ValueCell@0x15645578 -> 0x15645434
```

This trace shows three reference objects being enqueued. The location of the reference object and the referent is displayed, along with the class name of the object.

Note: If finalizer objects are listed in the trace, it does not mean that the corresponding finalizer has run. It means only that the finalizer has been queued in the finalizer thread.

-Xtgc:scavenger

This trace prints a histogram following each scavenger collection.

A graph is shown of the different classes of objects remaining in the survivor space, together with the number of occurrences of each class and the age of each object (the number of times it has been flipped). A sample of the output from a single scavenge is shown as follows:


```
{SCAV: tgcScavenger OBJECT HISTOGRAM}

{SCAV: | class | instances of age 0-14 in semi-space |
{SCAV: java/lang/ref/SoftReference 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/io/FileOutputStream 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: sun/nio/cs/StreamEncoder$ConverterSE 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/io/FileInputStream 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: char[] 0 102 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/lang/ref/SoftReference[] 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/io/BufferedOutputStream 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/io/BufferedWriter 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/io/OutputStreamWriter 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/io/PrintStream 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/io/BufferedInputStream 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/lang/Thread[] 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: java/lang/ThreadGroup[] 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: sun/io/ByteToCharCp1252 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
{SCAV: sun/io/CharToByteCp1252 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
```

-Xtgc:terse

This trace dumps the contents of the entire heap before and after a garbage collection.

This is an extremely large trace. For each object or free chunk in the heap, a line of trace output is produced. Each line contains the base address, "a" if it is an allocated object and "f" if it is a free chunk, the size of the chunk in bytes, and if it is an object, its class name. A sample is shown as follows:

```
*DH(1)* 230AD778 a x0000001C java/lang/String
*DH(1)* 230AD794 a x00000048 char[]
*DH(1)* 230AD7DC a x00000018 java/lang/StringBuffer
*DH(1)* 230AD7F4 a x00000030 char[]
*DH(1)* 230AD824 a x00000054 char[]
*DH(1)* 230AD878 a x0000001C java/lang/String
*DH(1)* 230AD894 a x00000018 java/util/HashMapEntry
*DH(1)* 230AD8AC a x0000004C char[]
*DH(1)* 230AD8F8 a x0000001C java/lang/String
*DH(1)* 230AD914 a x0000004C char[]
*DH(1)* 230AD960 a x00000018 char[]
*DH(1)* 230AD978 a x0000001C java/lang/String
*DH(1)* 230AD994 a x00000018 char[]
*DH(1)* 230AD9AC a x00000018 java/lang/StringBuffer
*DH(1)* 230AD9C4 a x00000030 char[]
*DH(1)* 230AD9F4 a x00000054 char[]
*DH(1)* 230ADA48 a x0000001C java/lang/String
*DH(1)* 230ADA64 a x00000018 java/util/HashMapEntry
*DH(1)* 230ADA7C a x00000050 char[]
*DH(1)* 230ADACC a x0000001C java/lang/String
*DH(1)* 230ADAE8 a x00000050 char[]
*DH(1)* 230ADB38 a x00000018 char[]
*DH(1)* 230ADB50 a x0000001C java/lang/String
*DH(1)* 230ADB6C a x00000018 char[]
*DH(1)* 230ADB84 a x00000018 java/lang/StringBuffer
*DH(1)* 230ADB9C a x00000030 char[]
*DH(1)* 230ADBCC a x00000054 char[]
*DH(1)* 230ADC20 a x0000001C java/lang/String
*DH(1)* 230ADC3C a x00000018 java/util/HashMapEntry
*DH(1)* 230ADC54 a x0000004C char[]
```

Finding which methods allocated large objects

You can use `-Xdump:stack:events=allocation,filter=#1k` to determine the source of large object allocations.

A stack trace can be generated to show which methods are responsible for allocating objects over a given size. The command-line option to use is:

```
-Xdump:stack:events=allocation,filter=#1k
```

This command prints stack information for all allocations over 1k. You can modify this value as required. However, the lower the value that is specified, the greater the affect on performance of the running application.

It is also possible to specify ranges of allocation sizes. For example, to print stack traces for allocations 2 - 4 Mb in size you can use:

```
-Xdump:stack:events=allocation,filter=#2m..4m
```

Omitting a valid filter produces the message: JVMDUMP036I Invalid or missing -Xdump filter

Sample output for the **-Xdump:stack:events=allocation,filter=#1k** option looks like:

```
./java "-Xdump:stack:events=allocation,filter=#1k" -version
JVMDUMP006I Processing dump event "allocation", detail "1264 bytes, class [B" - please wait.
Thread=main (088B9C4C) Status=Running
at java/lang/System.getPropertyList()[Ljava/lang/String; (Native Method)
at java/lang/System.ensureProperties()V (System.java:254)
at java/lang/System.<clinit>()V (System.java:101)
at java/lang/J9VMInternals.initializeImpl(Ljava/lang/Class;)V (Native Method)
at java/lang/J9VMInternals.initialize(Ljava/lang/Class;)V (J9VMInternals.java:200)
at java/lang/ClassLoader.initializeClassLoaders()V (ClassLoader.java:72)
at java/lang/Thread.initialize(ZLjava/lang/ThreadGroup;Ljava/lang/Thread;)V (Thread.java:325)
at java/lang/Thread.<init>(Ljava/lang/String;Ljava/lang/Object;IZ)V (Thread.java:124)
JVMDUMP013I Processed dump event "allocation", detail "1264 bytes, c lass [B".
```

Chapter 29. Class-loader diagnostics

There are some diagnostics that are available for class-loading.

The topics that are discussed in this chapter are:

- “Class-loader command-line options”
- “Class-loader runtime diagnostics”
- “Loading from native code” on page 348

Class-loader command-line options

There are some extended command-line options that are available

These options are:

-verbose:dynload

Provides detailed information as each class is loaded by the JVM, including:

- The class name and package.
- For class files that were in a .jar file, the name and directory path of the .jar (for bootstrap classes only).
- Details of the size of the class and the time taken to load the class.

The data is written out to stderr. An example of the output follows:

```
<Loaded java/lang/String from C:\sdk\jre\lib\vm.jar>
<Class size 17258; ROM size 21080; debug size 0>
<Read time 27368 usec; Load time 782 usec; Translate time 927 usec>
```

-Xfuture

Turns on strict class-file format checks. Use this flag when you are developing new code because stricter checks will become the default in future releases. By default, strict format checks are disabled.

-Xverify[:<option>]

With no parameters, enables the Java bytecode verifier, which is the default. Therefore, if used on its own with no parameters, the option has no effect. Optional parameters are:

- **all** - enable maximum verification
- **none** - disable the verifier
- **remote** - enables strict class-loading checks on remotely loaded classes

The verifier is on by default and must be enabled for all production servers. Running with the verifier off, is not a supported configuration. If you encounter problems and the verifier was turned off using **-Xverify:none**, remove this option and try to reproduce the problem.

Class-loader runtime diagnostics

Use the command-line parameter **-Dibm.cl.verbose=<class_expression>** to enable you to trace the way the class loaders find and load application classes.

For example:

```
C:\j9test>java -Dibm.cl.verbose=*HelloWorld hw.HelloWorld
```

produces output that is similar to this:

```
ExtClassLoader attempting to find hw.HelloWorld
ExtClassLoader using classpath C:\sdk\jre\lib\ext\CmpCrmf.jar;C:\sdk\jre\lib\ext\dtfj-interface.jar;
C:\sdk\jre\lib\ext\dtfj.jar;C:\sdk\jre\lib\ext\gskikm.jar;C:\sdk\jre\lib\ext\ibmcmsprovider.jar;C:\s
dk\jre\lib\ext\ibmjcefpis.jar;C:\sdk\jre\lib\ext\ibmjceprovider.jar;C:\sdk\jre\lib\ext\ibmkeycert.ja
r;C:\sdk\jre\lib\ext\IBMKeyManagementServer.jar;C:\sdk\jre\lib\ext\ibmpkcs11.jar;C:\sdk\jre\lib\ext\
ibmpkcs11impl.jar;C:\sdk\jre\lib\ext\ibmsaslprovider.jar;C:\sdk\jre\lib\ext\indicim.jar;C:\sdk\jre\l
ib\ext\jaccess.jar;C:\sdk\jre\lib\ext\JawBridge.jar;C:\sdk\jre\lib\ext\jdmview.jar
ExtClassLoader path element C:\sdk\jre\lib\ext\CmpCrmf.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\dtfj-interface.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\dtfj.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\gskikm.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\ibmcmsprovider.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\ibmjcefpis.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\ibmjceprovider.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\ibmkeycert.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\IBMKeyManagementServer.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\ibmpkcs11.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\ibmpkcs11impl.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\ibmsaslprovider.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\indicim.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\jaccess.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\JawBridge.jar does not exist
ExtClassLoader path element C:\sdk\jre\lib\ext\jdmview.jar does not exist
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\CmpCrmf.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\dtfj-interface.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\dtfj.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\gskikm.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\ibmcmsprovider.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\ibmjcefpis.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\ibmjceprovider.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\ibmkeycert.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\IBMKeyManagementServer.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\ibmpkcs11.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\ibmpkcs11impl.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\ibmsaslprovider.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\indicim.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\jaccess.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\JawBridge.jar
ExtClassLoader could not find hw/HelloWorld.class in C:\sdk\jre\lib\ext\jdmview.jar
ExtClassLoader could not find hw.HelloWorld

AppClassLoader attempting to find hw.HelloWorld
AppClassLoader using classpath C:\j9test
AppClassLoader path element C:\j9test does not exist
AppClassLoader found hw/HelloWorld.class in C:\j9test
AppClassLoader found hw.HelloWorld
```

The sequence of the loaders' output is a result of the “delegate first” convention of class loaders. In this convention, each loader checks its cache and then delegates to its parent loader. Then, if the parent returns null, the loader checks the file system or equivalent. This part of the process is reported in the example above.

The `<class_expression>` can be given as any Java regular expression. “Dic*” matches all classes with names begins with “Dic”, and so on.

Loading from native code

A class loader loads native libraries for a class.

Class loaders look for native libraries in different places:

- If the class that makes the native call is loaded by the Bootstrap Classloader, this loader looks in the 'sun.boot.library.path' to load the libraries.

- If the class that makes the native call is loaded by the Extensions Classloader, this loader looks in the 'java.ext.dirs' first, then 'sun.boot.library.path,' and finally the 'java.library.path', to load the libraries.
- If the class that makes the native call is loaded by the Application Classloader, this loader looks in the 'sun.boot.library.path', then the 'java.library.path', to load the libraries.
- If the class that makes the native call is loaded by a Custom Classloader, this loader defines the search path to load libraries.

Chapter 30. Shared classes diagnostics

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

For an introduction to shared classes, see Chapter 4, “Class data sharing,” on page 33.

The topics that are discussed in this chapter are:

- “Deploying shared classes”
- “Dealing with runtime bytecode modification” on page 356
- “Understanding dynamic updates” on page 358
- “Using the Java Helper API” on page 361
- “Understanding shared classes diagnostics output” on page 363
- “Debugging problems with shared classes” on page 366
- “Class sharing with OSGi ClassLoading framework” on page 370

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.

If the same user will always be using the same application, either use the default cache name (which includes the user name) or specify a cache name specific to the application. The user name can be incorporated into a cache name using the %u modifier, which causes each user running the application to get a separate cache.

On Linux, AIX, z/OS, and i5/OS platforms, if multiple users in the same operating system group are running the same application, use the **groupAccess** suboption, which creates the cache allowing all users in the same primary group to share the same cache. If multiple operating system groups are running the same application, the %g modifier can be added to the cache name, causing each group running the application to get a separate cache.

Multiple applications or different JVM installations can share the same cache provided that the JVM installations are of the same service release level. It is possible for different JVM service releases to share the same cache, but it is not advised. The JVM will attempt to destroy and re-create a cache created by a different service release. See “Compatibility between service releases” on page 354 for more information.

Small applications that load small numbers of application classes should all try to share the same cache, because they will still be able to share bootstrap classes. For

large applications that contain completely different classes, it might be more sensible for them to have a class cache each, because there will be few common classes and it is then easier to selectively clean up caches that aren't being used.

On Windows, caches are stored as memory-mapped files in the user's directory in "Documents and Settings". Therefore, one user creating a cache named "myCache" and another user creating a cache named "myCache" will cause two different caches named "myCache" to be created. Because the data is stored on disk, rather than in shared memory, there are fewer resource constraints on the number of caches that can be created.

On Linux, AIX, z/OS, and i5/OS, only one cache of each name can exist and different users must have permissions to access it.

Cache access

A JVM can access a shared class cache with either read-write or read-only access. Read-write access is the default and gives all users equal rights to update the cache. Use the **-Xshareclasses:readonly** option for read-only access.

Opening a cache as read-only makes it easier to administer operating system permissions. A cache created by one user cannot be opened read-write by other users, but other users can reduce startup time by opening the cache as read-only. Opening a cache as read-only also prevents corruption of the cache. This option can be useful on production systems where one instance of an application corrupting the cache might affect the performance of all other instances.

When a cache is opened read-only, class files of the application that are modified or moved cannot be updated in the cache. Sharing is disabled for the modified or moved containers for that JVM.

Cache housekeeping

Unused caches on a system waste resources that might be used by another application. Ensuring that caches are sensibly managed is important.

The **destroy** and **destroyAll** utilities are used to explicitly remove named caches or all caches on a system. However, the **expire=<time>** suboption is useful for automatically clearing out old caches that have not been used for some time. The suboption is added to the **-Xshareclasses** option. The parameter **<time>** is measured in minutes. This option causes the JVM to scan automatically for caches that have not been connected to for a period greater than or equal to **<time>**, before initializing the shared classes support. A command line with **-Xshareclasses:name=myCache,expire=10000** automatically removes any caches that have been unused for a week, before creating or connecting to myCache. Setting **expire=0** removes all existing caches before initializing the shared classes support, so that a fresh cache is always created.

On Linux, AIX, z/OS, and i5/OS platforms, the limitation with this housekeeping is that caches can be removed only by the user who created them, or by root. If the `javasharedresources` directory in `/tmp` is accidentally deleted, the control files that allow JVMs to identify shared memory areas are lost. Therefore the shared memory areas are also lost, although they still exist in memory. The next JVM to start after `javasharedresources` is deleted attempts to identify and remove any such lost memory areas before recreating `javasharedresources`. The JVM can remove only memory areas that were created by the user running that JVM, unless the JVM is running as root. For more information, see the description of a hard reset in

“Dealing with initialization problems” on page 367. Rebooting a system causes all shared memory to be lost, although the control files still exist in `javasharedresources`. The existing control files are deleted or reused. The shared memory areas are re-created next time the JVM starts.

Cache performance

Shared classes use optimizations to maintain performance under most circumstances. However, there are configurable factors that can affect shared classes performance.

Use of Java archive and compressed files

The cache keeps itself up-to-date with file system updates by constantly checking file system timestamps against the values in the cache.

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

`.class` files can be created or deleted from a directory at any time. If you include a directory name in a classpath, shared classes performance can be affected because the directory is constantly checked for classes. The impact on performance might be greater if the directory name is near the beginning of the classpath string. For example, consider a classpath of `/dir1:jar1.jar:jar2.jar:jar3.jar;`. When loading any class from the cache using this classpath, the directory `/dir1` must be checked for the existence of the class for every class load. This checking also requires fabricating the expected directory from the package name of the class. This operation can be expensive.

Advantages of not filling the cache

A full shared classes cache is not a problem for any JVMs connected to it. However, a full cache can place restrictions on how much sharing can be performed by other JVMs or applications.

ROMClasses are added to the cache and are all unique. Metadata is added describing the ROMClasses and there can be multiple metadata entries corresponding to a single ROMClass. For example, if class A is loaded from `myApp1.jar` and another JVM loads the same class A from `myOtherApp2.jar`, only one ROMClass exists in the cache. However there are two pieces of metadata that describe the source locations.

If many classes are loaded by an application and the cache is 90% full, another installation of the same application can use the same cache. The extra information that must be added about the classes from the second application is minimal.

After the extra metadata has been added, both installations can share the same classes from the same cache. However, if the first installation fills the cache completely, there is no room for the extra metadata. The second installation cannot share classes because it cannot update the cache. The same limitation applies for classes that become stale and are redeemed. See “Redeeming stale classes” on page 360. 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.

Very long classpaths

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

Matching very long classpaths is initially expensive, but successful and failed matches are remembered. Therefore, loading classes from the cache using very long classpaths is much faster than loading from disk.

Growing classpaths

Where possible, avoid gradually growing a classpath in a `URLClassLoader` using `addURL()`. Each time an entry is added, an entire new classpath must be added to the cache.

For example, if a classpath with 50 entries is grown using `addURL()`, you might create 50 unique classpaths in the cache. This gradual growth uses more cache space and has the potential to slow down classpath matching when loading classes.

Concurrent access

A shared class cache can be updated and read concurrently by any number of JVMs. Any number of JVMs can read from the cache while a single JVM is writing to it.

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

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

Class GC with shared classes

Running with shared classes has no affect on class garbage collection. Classloaders loading classes from the shared class cache can be garbage collected in the same way as classloaders that load classes from disk. If a classloader is garbage collected, the `ROMClasses` it has added to the cache persist.

Compatibility between service releases

Use the most recent service release of a JVM for any application.

It is not recommended for different service releases to share the same class cache concurrently. A class cache is compatible with earlier and later service releases. However, there might be small changes in the class files or the internal class file format between service releases. These changes might result in duplication of classes in the cache. For example, a cache created by a given service release can continue to be used by an updated service release, but the updated service release might add extra classes to the cache if space allows.

To reduce class duplication, if the JVM connects to a cache which was created by a different service release, it attempts to destroy the cache then re-create it. This automated housekeeping feature is designed so that when a new JVM level is used with an existing application, the cache is automatically refreshed. However, the refresh only succeeds if the cache is not in use by any other JVM. If the cache is in use, the JVM cannot refresh the cache, but uses it where possible.

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.

You might need to do a manual cleanup when you see the following messages:

```
JVMSHRC020E An error has occurred while opening semaphore
JVMSHRC017E Error code: -308
JVMSHRC320E Error recovery: destroying shared memory semaphores.
JVMJ9VM015W Initialization error for library j9shr24(11):
JVMJ9VM009E J9VMD1Main failed
```

In response to these messages, run the following command as root:

```
ipcs -a
```

Record the System V memory and semaphore IDs using these rules:

- For Java 5 SR11 and later, record all semaphores IDs with corresponding keys having a "Most Significant Byte" (MSB) in the range 0x41 to 0x54.
- For Java 5 SR11 and later, record all memory IDs with corresponding keys having an MSB in the range 0x21 to 0x34.
- For earlier versions of Java 5, do the same by recording all semaphore IDs and all memory IDs, where the corresponding keys begin with an MSB in the range 0x01 to 0x14.

For each System V semaphore ID recorded, use the following command to delete the semaphore:

```
ipcrm -s <semid>
```

where <semid> is the System V semaphore ID.

For each System V shared memory ID recorded, use the following command to delete the shared memory:

```
ipcrm -m <shmid>
```

where <shmid> is the System V shared memory ID.

Dealing with runtime bytecode modification

Modifying bytecode at runtime is an increasingly popular way to engineer required function into classes. Sharing modified bytecode improves startup time, especially when the modification being used is expensive. You can safely cache modified bytecode and share it between JVMs, but there are many potential problems because of the added complexity. It is important to understand the features described in this section to avoid any potential problems.

This section contains a brief summary of the tools that can help you to share modified bytecode.

Potential problems with runtime bytecode modification

The sharing of modified bytecode can cause potential problems.

When a class is stored in the cache, the location from which it was loaded and a time stamp indicating version information are also stored. When retrieving a class from the cache, the location from which it was loaded and the time stamp of that location are used to determine whether the class should be returned. The cache does not note whether the bytes being stored were modified before they were defined unless it is specifically told so. Do not underestimate the potential problems that this modification could introduce:

- In theory, unless all JVMs sharing the same classes are using exactly the same bytecode modification, JVMs could load incorrect bytecode from the cache. For example, if JVM1 populates a cache with modified classes and JVM2 is not using a bytecode modification agent, but is sharing classes with the same cache, it could incorrectly load the modified classes. Likewise, if two JVMs start at the same time using different modification agents, a mix of classes could be stored and both JVMs will either throw an error or demonstrate undefined behavior.
- An important prerequisite for caching modified classes is that the modifications performed must be deterministic and final. In other words, an agent which performs a particular modification under one set of circumstances and a different modification under another set of circumstances, cannot use class caching. This is because only one version of the modified class can be cached for any given agent and once it is cached, it cannot be modified further or returned to its unmodified state.

In practice, modified bytecode can be shared safely if the following criteria are met:

- Modifications made are deterministic and final (described above).
- The cache knows that the classes being stored are modified in a particular way and can partition them accordingly.

The VM provides features that allow you to share modified bytecode safely, for example using "modification contexts". However, if a JVMTI agent is unintentionally being used with shared classes without a modification context, this

usage does not cause unexpected problems. In this situation, if the VM detects the presence of a JVMTI agent that has registered to modify class bytes, it forces all bytecode to be loaded from disk and this bytecode is then modified by the agent. The potentially modified bytecode is passed to the cache and the bytes are compared with known classes of the same name. If a matching class is found, it is reused; otherwise, the potentially modified class is stored in such a way that other JVMs cannot load it accidentally. This method of storing provides a "safety net" that ensures that the correct bytecode is always loaded by the JVM running the agent, but any other JVMs sharing the cache will be unaffected. Performance during class loading could be affected because of the amount of checking involved, and because bytecode must always be loaded from disk. Therefore, if modified bytecode is being intentionally shared, the use of modification contexts is recommended.

Modification contexts

A modification context creates a private area in the cache for a given context, so that multiple copies or versions of the same class from the same location can be stored using different modification contexts. You choose the name for a context, but it must be consistent with other JVMs using the same modifications.

For example, one JVM uses a JVMTI agent "agent1", a second JVM uses no bytecode modification, a third JVM also uses "agent1", and a fourth JVM uses a different agent, "agent2". If the JVMs are started using the following command lines (assuming that the modifications are predictable as described above), they should all be able to share the same cache:

```
java -agentlib:agent1 -Xshareclasses:name=cache1,modified=myAgent1 myApp.ClassName
java -Xshareclasses:name=cache1 myApp.ClassName
java -agentlib:agent1 -Xshareclasses:name=cache1,modified=myAgent1 myApp.ClassName
java -agentlib:agent2 -Xshareclasses:name=cache1,modified=myAgent2 myApp.ClassName
```

SharedClassHelper partitions

Modification contexts cause all classes loaded by a particular JVM to be stored in a separate cache area. If you need a more granular approach, the SharedClassHelper API can store individual classes under "partitions".

This ability to use partitions allows an application class loader to have complete control over the versioning of different classes and is particularly useful for storing bytecode woven by Aspects. A partition is a string key used to identify a set of classes. For example, a system might weave a number of classes using a particular Aspect path and another system might weave those classes using a different Aspect path. If a unique partition name is computed for the different Aspect paths, the classes can be stored and retrieved under those partition names.

The default application class loader or bootstrap class loader does not support the use of partitions; instead, a SharedClassHelper must be used with a custom class loader.

Using the safemode option

If you have unexpected results or VerifyErrors from cached classes, use safemode to determine if the bytecode from the cache is correct for your JVM.

Unexpected results from cached classes, or VerifyErrors, might be caused by the wrong classes being returned. Another cause might be incorrect cached classes. You can use a debugging mode called safemode to find whether the bytecode being loaded from the cache is correct for the JVM you are using.

safemode is a suboption of **-Xshareclasses**. It prevents the use of shared classes. safemode does not add classes to a cache.

When you use safemode with a populated cache, it forces the JVM to load all classes from disk and then apply any modifications to those classes. The class loader then tries to store the loaded classes in the cache. The class being stored is compared byte-for-byte against the class that would be returned if the class loader had not loaded the class from disk. If any bytes do not match, the mismatch is reported to stderr. Using safemode helps ensure that all classes are loaded from disk. safemode provides a useful way of verifying whether the bytes loaded from the shared class cache are the expected bytes.

Do not use safemode in production systems, because it is only a debugging tool and does not share classes.

Further considerations for runtime bytecode modification

There are a number of additional items that you need to be aware of when using the cache with runtime bytecode modification.

If bytecode is modified by a non-JVMTI agent and defined using the JVM's application classloader when shared classes are enabled, these modified classes are stored in the cache and nothing is stored to indicate that these are modified classes. Another JVM using the same cache will therefore load the classes with these modifications. If you are aware that your JVM is storing modified classes in the cache using a non-JVMTI agent, you are advised to use a modification context with that JVM to protect other JVMs from the modifications.

Combining partitions and modification contexts is possible but not recommended, because you will have "partitions inside partitions". In other words, a partition A stored under modification context X will be different from partition A stored under modification context B.

Because the shared class cache is a fixed size, storing many different versions of the same class might require a much larger cache than the size that is typically required. However, note that the identical classes are never duplicated in the cache, even across modification contexts or partitions. Any number of metadata entries might describe the class and where it came from, but they all point to the same class bytes.

If an update is made to the file system and the cache marks a number of classes as stale as a result, note that it will mark all versions of each class as stale (when versions are stored under different modification contexts or partitions) regardless of the modification context being used by the JVM that caused the classes to be marked stale.

Understanding dynamic updates

The shared class cache must respond to file system updates; otherwise, a JVM might load classes from the cache that are out of date or "stale". After a class has been marked stale, it is not returned by the cache if it is requested by a class loader. Instead, the class loader must reload the class from disk and store the updated version in the cache.

The cache is managed in a way that helps ensure that the following challenges are addressed:

- Java archive and compressed files are usually locked by class loaders when they are in use. The files can be updated when the JVM shuts down. Because the cache persists beyond the lifetime of any JVM using it, subsequent JVMs connecting to the cache check for Java archive and compressed file updates.
- .class files that are not in a .jar file can be updated at any time during the lifetime of a JVM. The cache checks for individual class file updates.
- .class files can be created or removed from directories found in classpaths at any time during the lifetime of a JVM. The cache checks the classpath for classes that have been created or removed.
- .class files must be in a directory structure that reflects their package structure. This structure helps ensure that when checking for updates, the correct directories are searched.

Class files contained in jars and compressed files, and class files stored as .class files on the file system, are accessed and used in different ways. The result is that the cache treats them as two different types. Updates are managed by writing file system time stamps into the cache.

Classes found or stored using a SharedClassTokenHelper cannot be maintained in this way, because Tokens are meaningless to the cache.

Storing classes

When a classpath is stored in the cache, the Java archive and compressed files are time stamped. These time stamps are stored as part of the classpath. Directories are not time stamped. When a ROMClass is stored, if it came from a .class file on the file system, the .class file it came from is time stamped and this time stamp is stored. Directories are not time stamped because there is no guarantee that subsequent updates to a file cause an update to the directory holding the file.

If a compressed or Java archive file does not exist, the classpath containing it can still be added to the cache, but ROMClasses from this entry are not stored. If an attempt is made to add a ROMClass to the cache from a directory, but the ROMClass does not exist as a .class file, it is not stored in the cache.

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

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

Finding classes

When the JVM finds a class in the cache, it must make more checks than when it stores a class.

When a potential match has been found, if it is a .class file on the file system, the time stamps of the .class file and the ROMClass stored in the cache are compared. Regardless of the source of the ROMClass (.jar or .class file), every Java archive and compressed file entry in the calling classpath, up to and including the index at which the ROMClass was “found”, must be checked for updates by

obtaining the time stamps. Any update might mean that another version of the class being returned had already been added earlier in the classpath.

Additionally, any classpath entries that are directories might contain `.class` files that “shadow” the potential match that has been found. Class files might be created or deleted in these directories at any point. Therefore, when the classpath is walked and jars and compressed files are checked, directory entries are also checked to see whether any `.class` files have been created unexpectedly. This check involves building a string by using the classpath entry, the package names, and the class name, and then looking for the class file. This procedure is expensive if many directories are being used in class paths. Therefore, using jar files gives better shared classes performance.

Marking classes as stale

When an individual `.class` file is updated, only the class or classes stored from that `.class` file are marked “stale”.

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

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

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

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

Classes in the cache that have been loaded from **c**, **d**, and **a** are marked stale. Making a single update to one jar file might cause many classes in the cache to be marked stale. To avoid massive duplication as classes are updated, stale classes can be marked as not stale, or “redeemed”, if it is proved that they are not in fact stale.

Redeeming stale classes

Because classes are marked stale when a class path update occurs, many of the classes marked stale might not have updated. When a class loader stores a class, and in doing so effectively “updates” a stale class, you can “redeem” the stale class if you can prove that it has not in fact changed.

For example, assume that class **X** is stored in a cache after obtaining it from location **c**, where **c** is part of the classpath **a;b;c;d**. Suppose **a** is updated. The update means that **a** might now contain a new version of class **X**. For this example, assume **a** does not contain a new version of class **X**. The update marks all classes loaded from **b**, **c**, and **d** as stale. Next, another JVM must load class **X**. The JVM asks the cache for class **X**, but it is stale, so the cache does not return the class. Instead, the class loader fetches class **X** from disk and stores it in the cache, again using classpath **a;b;c;d**. The cache checks the loaded version of **X** against the stale version of **X** and, if it matches, the stale version is “redeemed”.

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 and Javadoc information for these classes is shipped with the SDK (some of which is reproduced here).

com.ibm.oti.shared.Shared

The `Shared` class contains static utility methods:

`getSharedClassHelperFactory()` and `isSharingEnabled()`. If **-Xshareclasses** is specified on the command line and sharing has been successfully initialized, `isSharingEnabled()` returns true. If sharing is enabled, `getSharedClassHelperFactory()` 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 classpath, that load classes from multiple locations.
- `SharedClassURLClasspathHelper`. Store and find classes using a classpath of URLs. For use by class loaders that load classes using a URL class path

Compatibility between Helpers is as follows: Classes stored by `SharedClassURLHelper` can be found using a `SharedClassURLClasspathHelper` and the opposite also applies. However, classes stored using a `SharedClassTokenHelper` can be found only by using a `SharedClassTokenHelper`.

Note: Classes stored using the URL Helpers are updated dynamically by the cache (see “Understanding dynamic updates” on page 358). 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.

For a detailed description of each helper and how to use it, see the Javadoc information shipped with the SDK.

com.ibm.oti.shared.SharedClassStatistics

The SharedClassStatistics class provides static utilities that return the total cache size and the amount of free bytes in the cache.

SharedClassHelper API

The SharedClassHelper API provides functions to find and store shared classes.

These functions are:

findSharedClass

Called after the class loader has asked its parent for a class, but before it has looked on disk for the class. If findSharedClass returns a class (as a byte[]), pass this class to defineClass(), which defines the class for that JVM and return it as a java.lang.Class object. The byte[] returned by findSharedClass is not the actual class bytes. The effect is that you cannot monitor or manipulate the bytes in the same way as class bytes loaded from a disk. If a class is not returned by findSharedClass, the class is loaded from disk (as in the nonshared case) and then the java.lang.Class defined is passed to storeSharedClass.

storeSharedClass

Called if the class loader has loaded class bytes from disk and has defined them using defineClass. Do not use storeSharedClass to try to store classes that were defined from bytes returned by findSharedClass.

setSharingFilter

Register a filter with the SharedClassHelper. The filter is used to decide which classes are found and stored in the cache. Only one filter can be registered with each SharedClassHelper.

You must resolve how to deal with metadata that cannot be stored. An example is when java.security.CodeSource or java.util.jar.Manifest objects are derived from jar files. For each jar, the best way to deal with metadata that cannot be stored is always to load the first class from the jar. Load the class regardless of whether it exists in the cache or not. This load activity initializes the required metadata in the class loader, which can then be cached internally. When a class is then returned by findSharedClass, the function indicates where the class has been loaded from. The result is that the correct cached metadata for that class can be used.

It is not incorrect usage to use storeSharedClass to store classes that were loaded from disk, but which are already in the cache. The cache sees that the class is a duplicate of an existing class, it is not duplicated, and so the class continues to be shared. However, although it is handled correctly, a class loader that uses only storeSharedClass is less efficient than one that also makes appropriate use of findSharedClass.

Understanding shared classes diagnostics output

When running in shared classes mode, a number of diagnostics tools can help you. The verbose options are used at runtime to show cache activity and you can use the `printStats` and `printAllStats` utilities to analyze the contents of a shared class cache.

This section tells you how to interpret the output.

Verbose output

The **verbose** suboption of **-Xshareclasses** gives the most concise and simple diagnostic output on cache usage.

See “JVM command-line options” on page 444. Verbose output will typically look like this:

```
>java -Xshareclasses:name=myCache,verbose -Xscmx10k HelloWorld
[-Xshareclasses verbose output enabled]
JVMSHRC158I Successfully created shared class cache "myCache"
JVMSHRC166I Attached to cache "myCache", size=10200 bytes
JVMSHRC096I WARNING: Shared Cache "myCache" is full. Use -Xscmx to set cache size.
Hello
JVMSHRC168I Total shared class bytes read=0. Total bytes stored=9284
```

This output shows that a new cache called `myCache` was created, which was only 10 kilobytes in size and the cache filled up almost immediately. The message displayed on shut down shows how many bytes were read or stored in the cache.

VerboseIO output

The `verboseIO` output is far more detailed, and is used at run time to show classes being stored and found in the cache.

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

```
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 0... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 3... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 17... Failed.
Storing class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 17... Succeeded.
```

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

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

```
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 0... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 3... Failed.
Finding class org/eclipse/ui/internal/UIWorkspaceLock in shared cache for cldr id 17... Succeeded.
```

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

VerboseHelper output

You can also obtain diagnostics from the Java SharedClassHelper API using the **verboseHelper** suboption.

The output is divided into information messages and error messages:

- Information messages are prefixed with:
Info for SharedClassHelper id <n>: <message>
- Error messages are prefixed with:
Error for SharedClassHelper id <n>: <message>

Use the Java Helper API to obtain this output; see “Using the Java Helper API” on page 361.

printStats utility

The **printStats** utility prints summary information about the specified cache to the standard error output.

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

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

```
baseAddress      = 0x20EE0058
endAddress       = 0x222DFFF8
allocPtr         = 0x21841AF8

cache size       = 20971432
free bytes       = 10992796
ROMClass bytes   = 9837216
Metadata bytes   = 141420
Metadata % used  = 1%

# ROMClasses     = 2167
# Classpaths     = 16
# URLs           = 0
# Tokens         = 0
# Stale classes  = 3
% Stale classes  = 0%
```

Cache is 47% full

baseAddress and endAddress

Give the boundary addresses of the shared memory area containing the classes.

allocPtr

Is the address where ROMClass data is currently being allocated in the cache.

cache size and free bytes

cache size shows the total size of the shared memory area in bytes, and free bytes shows the free bytes remaining.

ROMClass bytes

Is the number of bytes of class data in the cache.

Metadata bytes

Is the number of bytes of non-class data that describe the classes.

Metadata % used

Shows the proportion of metadata bytes to class bytes; this proportion indicates how efficiently cache space is being used.

ROMClasses

Indicates the number of classes in the cache. The cache stores ROMClasses (the class data itself, which is read-only) and it also stores information about the location from which the classes were loaded. This information is stored in different ways, depending on the Java SharedClassHelper API used to store the classes. For more information, see "Using the Java Helper API" on page 361.

Classpaths, URLs, and Tokens

Indicates the number of classpaths, URLs, and tokens in the cache. Classes stored from a SharedClassLoaderHelper are stored with a Classpath. Classes stored using a SharedClassLoaderHelper are stored with a URL. Classes stored using a SharedClassTokenHelper are stored with a Token. Most classloaders, including the bootstrap and application classloaders, use a SharedClassLoaderHelper. The result is that it is most common to see Classpaths in the cache.

The number of Classpaths, URLs, and Tokens stored is determined by a number of factors. For example, every time an element of a Classpath is updated, such as when a .jar file is rebuilt, a new Classpath is added to the cache. Additionally, if "partitions" or "modification contexts" are used, they are associated with the Classpath, URL, or Token. A Classpath, URL, or Token is stored for each unique combination of partition and modification context.

Stale classes

Are classes that have been marked as "potentially stale" by the cache code, because of an operating system update. See "Understanding dynamic updates" on page 358.

% Stale classes

Is an indication of the proportion of classes in the cache that have become stale.

printAllStats utility

The printAllStats utility is a suboption of **-Xshareclasses**, optionally taking a cache name using **name=<name>**. This utility lists the cache contents in order, providing as much diagnostic information as possible. Because the output is listed in chronological order, you can interpret it as an "audit trail" of cache updates. Because it is a cache utility, the JVM displays the information about the cache specified or the default cache and then exits.

Each JVM that connects to the cache receives a unique ID. Each entry in the output is preceded by a number indicating the JVM that wrote the data.

Classpaths

```
1: 0x2234FA6C CLASSPATH
   C:\myJVM\jdk\jre\lib\vm.jar
   C:\myJVM\jdk\jre\lib\core.jar
   C:\myJVM\jdk\jre\lib\charsets.jar
   C:\myJVM\jdk\jre\lib\graphics.jar
   C:\myJVM\jdk\jre\lib\security.jar
   C:\myJVM\jdk\jre\lib\ibmpkcs.jar
   C:\myJVM\jdk\jre\lib\ibmorb.jar
   C:\myJVM\jdk\jre\lib\ibmcfw.jar
   C:\myJVM\jdk\jre\lib\ibmorbapi.jar
```

```
C:\myJVM\jdk\jre\lib\ibmjcefw.jar
C:\myJVM\jdk\jre\lib\ibmjgssprovider.jar
C:\myJVM\jdk\jre\lib\ibmjsseprovider2.jar
C:\myJVM\jdk\jre\lib\ibmjaaslm.jar
C:\myJVM\jdk\jre\lib\ibmjaasactivelm.jar
C:\myJVM\jdk\jre\lib\ibmcertpathprovider.jar
C:\myJVM\jdk\jre\lib\server.jar
C:\myJVM\jdk\jre\lib\xml.jar
```

This output indicates that JVM 1 caused a class path to be stored at address 0x2234FA6C in the cache. The class path contains 17 entries, which are listed. If the class path was stored using a given partition or modification context, this information is also displayed.

ROMClasses

```
1: 0x2234F7DC ROMCLASS: java/lang/Runnable at 0x213684A8
   Index 1 in class path 0x2234FA6C
```

This output indicates that JVM 1 stored a class called java/lang/Runnable in the cache. The metadata about the class is stored at address 0x2234F7DC, and the class itself is written to address 0x213684A8. The output also indicates the class path against which the class is stored, and from which index in that class path the class was loaded. In the example, the class path is the same address as the one listed above. If a class is stale, it has !STALE! appended to the entry..

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.

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 Chapter 25, “Tracing Java applications and the JVM,” on page 283 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 358, if a class has been marked as “stale”, it is not returned by the cache.

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 class does not exist on the file system

The class might be sourced from a URL location that is not a file.

Why classes might not be found or stored

Safemode is being used

Classes are not found or stored in the cache in safemode. This behavior is expected for shared classes. See “Using the safemode option” on page 357.

The cache is corrupted

In the unlikely event that the cache is corrupted, no classes can be found or stored.

A SecurityManager is being used and the permissions have not been granted to the class loader

SharedClassPermissions must be granted to application class loaders so that they can share classes when a SecurityManager is used. For more information, see the SDK and Runtime guide for your platform.

Dealing with initialization problems

Shared classes initialization requires a number of operations to succeed. A failure might have many potential causes, and it is difficult to provide detailed message information following an initialization failure. Some common reasons for failure are listed here.

If you cannot see why initialization has failed from the command-line output, look at level 1 trace for more information regarding the cause of the failure. The *SDK and Runtime User Guide* for your platform provides detailed information about operating system limitations. A brief summary of potential reasons for failure is provided here.

Writing data into the javasharedresources directory

To initialize any cache, data must be written into a javasharedresources directory, which is created by the first JVM that needs it.

On Linux, AIX, z/OS, and i5/OS this directory is /tmp/javasharedresources, and is used only to store small amounts of metadata that identify the semaphore and shared memory areas. On Windows, this directory is C:\Documents and Settings\<username>\Local Settings\Application Data\javasharedresources. The memory-mapped file is written here..

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.

On Linux, AIX, z/OS, and i5/OS, caches are created with user-only access by default. Two users cannot share the same cache unless the **-Xshareclasses:groupAccess** command-line option is used when the cache is created. If user A creates a cache using **-Xshareclasses:name=myCache** and user B also tries to run the same command line, a failure occurs. The failure is because user B does not have permissions to access "myCache". Caches can be removed only by the user who created them, even if **-Xshareclasses:groupAccess** is used.

Initializing a cache

Non-persistent caches are the default on AIX and z/OS.

The following operations must succeed to initialize a cache:

1) Create a shared memory area

Possible problems depend on your platform.

Windows

A memory-mapped file is created on the file system and deleted when the operating system is restarted. The main reasons for failing to create a shared memory area are lack of available disk space and incorrect file write permissions.

Linux, AIX, z/OS, and i5/OS

The **SHMMAX** operating system environment variable by default is set low. **SHMMAX** limits the size of shared memory segment that can be allocated. If a cache size greater than **SHMMAX** is requested, the JVM attempts to allocate **SHMMAX** and outputs a message indicating that **SHMMAX** should be increased. For this reason, the default cache size is 16 MB.

z/OS Before using shared classes on z/OS, you must check in the *z/OS SDK and Runtime Environment User Guide* for APARs that must be installed. Also, check the operating system environment variables, as detailed in the user guide. On z/OS, the requested cache sizes are deliberately rounded to the nearest megabyte.

2) Create a shared semaphore

Shared semaphores are created in the `javasharedresources` directory. You must have write access to this directory.

3) Write metadata

Metadata is written to the `javasharedresources` directory. You must have write access to this directory.

If you are experiencing considerable initialization problems, try a hard reset:

1. Run `java -Xshareclasses:destroyAll` to remove all known memory areas and semaphores. On a Linux, AIX, or z/OS system, run this command as root, or as a user with `*ALLOBJ` authority on i5/OS.
2. Delete the `javasharedresources` directory and all of its contents.

3. On Linux, AIX, z/OS, or i5/OS the memory areas and semaphores created by the JVM might not have been removed using **-Xshareclasses:destroyAll**. This problem is addressed the next time you start the JVM. If the JVM starts and the `javasharedresources` directory does not exist, an automated cleanup is triggered. Any remaining shared memory areas that are shared class caches are removed. Follow one of these steps to reset the system and force the JVM to re-create the `javasharedresources` directory:
 - On Linux, AIX, or z/OS, using root authority, start the JVM with **-Xshareclasses**.
 - On i5/OS, using a user that has *ALLOBJ authority, start the JVM with **-Xshareclasses**.

Dealing with verification problems

Verification problems (typically seen as `java.lang.VerifyErrors`) are potentially caused by the cache returning incorrect class bytes.

This problem should not occur under typical usage, but there are two situations in which it could happen:

- The classloader is using a `SharedClassTokenHelper` and the classes in the cache are out-of-date (dynamic updates are not supported with a `SharedClassTokenHelper`).
- Runtime bytecode modification is being used that is either not fully predictable in the modifications it does, or it is sharing a cache with another JVM that is doing different (or no) modifications. Regardless of the reason for the `VerifyError`, running in `safemode` (see “Using the `safemode` option” on page 357) should show if any bytecode in the cache is inconsistent with what the JVM is expecting. When you have determined the cause of the problem, destroy the cache, correct the cause of the problem, and try again.

Dealing with cache problems

The following list describes possible cache problems.

Cache is full

A full cache is not a problem; it just means that you have reached the limit of data that you can share. Nothing can be added or removed from that cache and so, if it contains a lot of out-of-date classes or classes that are not being used, you must destroy the cache and create a new one.

Cache is corrupt

In the unlikely event that a cache is corrupt, no classes can be added or read from the cache and you must destroy the cache. A message is sent to `stderr` if the cache is corrupt. If a cache is corrupted during normal operation, all JVMs output the message and are forced to load all subsequent classes locally (not into the cache). The cache is designed to be resistant to crashes, so, if a JVM crash occurs during a cache update, the crash should not cause data to be corrupted.

Could not create the Java virtual machine message from utilities

This message does not mean that a failure has occurred. Because the cache utilities currently use the JVM launcher and they do not start a JVM, this message is always produced by the launcher after a utility has run. Because the JNI return code from the JVM indicates that a JVM did not start, it is an unavoidable message.

-Xscmx is not setting the cache size

You can set the cache size only when the cache is created because the size

is fixed. Therefore, **-Xscmx** is ignored unless a new cache is being created. It does not imply that the size of an existing cache can be changed using the parameter.

Class sharing with OSGi ClassLoading framework

Eclipse releases after 3.0 use the OSGi ClassLoading framework, which cannot automatically share classes. A Class Sharing adapter has been written specifically for use with OSGi, which allows OSGi classloaders to access the class cache.

Chapter 31. Using the Reliability, Availability, and Serviceability Interface

The JVM Reliability, Availability, and Serviceability Interface (JVMRI) allows an agent to access reliability, availability, and serviceability (RAS) functions by using a structure of pointers to functions.

The JVMRI interface will be deprecated in the near future and replaced by JVMTI extensions.

You can use the JVMRI interface to:

- Determine the trace capability that is present
- Set and intercept trace data
- Produce various dumps
- Inject errors

To use the JVMRI you must be able to build a native library, add the code for JVMRI callbacks (described below), and interface the code to the JVM through the JNI. This section provides the callback code but does not provide the other programming information.

This chapter describes the JVMRI in:

- “Preparing to use JVMRI”
- “JVMRI functions” on page 374
- “API calls provided by JVMRI” on page 375
- “RasInfo structure” on page 382
- “RasInfo request types” on page 382
- “Intercepting trace data” on page 382
- “Formatting” on page 383

Preparing to use JVMRI

Trace and dump functions in the JVMRI require the JVM trace and dump libraries to be loaded. These libraries will be loaded by default, but JVMRI will fail with a warning message if you specify **-Xtrace:none** or **-Xdump:none**.

See Appendix D, “Command-line options,” on page 439 for more information.

Writing an agent

This piece of code demonstrates how to write a very simple JVMRI agent.

When an agent is loaded by the JVM, the first thing that gets called is the entry point routine `JVM_OnLoad()`. Therefore, your agent must have a routine called `JVM_OnLoad()`. This routine then must obtain a pointer to the JVMRI function table. This is done by making a call to the `GetEnv()` function.

```
/* jvmri - jvmri agent source file. */
```

```
#include "jni.h"
#include "jvmri.h"
```

```

DgRasInterface *jvmri_intf = NULL;

JNIEXPORT jint JNICALL
JVM_OnLoad(JavaVM *vm, char *options, void *reserved)
{
    int rc;
    JNIEnv *env;

    /*
     * Get a pointer to the JNIEnv
     */

    rc = (*vm)->GetEnv(vm, (void **)&env, JNI_VERSION_1_2);
    if (rc != JNI_OK) {
        fprintf(stderr, "RASplugin001 Return code %d obtaining JNIEnv\n", rc);
        fflush(stderr);
        return JNI_ERR;
    }

    /*
     * Get a pointer to the JVMRI function table
     */

    rc = (*vm)->GetEnv(vm, (void **)&jvmri_intf, JVMRAS_VERSION_1_3);
    if (rc != JNI_OK) {
        fprintf(stderr, "RASplugin002 Return code %d obtaining DgRasInterface\n", rc);
        fflush(stderr);
        return JNI_ERR;
    }

    /*
     * Now a pointer to the function table has been obtained we can make calls to any
     * of the functions in that table.
     */

    .....

    return rc;
}

```

Registering a trace listener

Before you start using the trace listener, you must set the **-Xtrace** option with the relevant **external=tp_spec** information. This action tells the object which tracepoints to listen for.

See Appendix D, “Command-line options,” on page 439 for more information.

An agent can register a function that is called back when the JVM makes a trace point. The following example shows a trace listener that only increments a counter each time a trace point is taken.

```

void JNICALL
listener (
    void *env,
    void ** tl,
    const char *moduleName,
    unsigned int traceId,
    const char * format,
    va_list var )
{
    int *counter;

    if (*tl == NULL) {
        fprintf(stderr, "RASplugin100 first tracepoint for thread %p\n", env);
        *tl = (void *)malloc(4);
    }
}

```

```

        counter = (int *)*tl;
        *counter = 0;
    }

    counter = (int *)*tl;

    (*counter)++;

    fprintf(stderr, "Trace point total = %d\n", *counter );
}

```

Add this code to the `JVM_Onload()` function or a function that `JVM_Onload()` calls.

The following example is used to register the trace listener.

```

/*
 * Register the trace listener
 */

rc = jvmri_intf->TraceRegister50( env, listener );
if ( rc != JNI_OK )
{
    fprintf( stderr, "RASplugin003 Return code %d registering listener\n", rc );
    fflush( stderr );
    return JNI_ERR;
}

```

You can also do more difficult tasks with a trace listener, including formatting, displaying, and recording trace point information.

Changing trace options

This example uses the `TraceSet()` function to change the JVM trace setting. It makes the assumption that the options string that is specified with the **-Xrun** option and passed to `JVM_Onload()` is a trace setting.

```

/*
 * If an option was supplied, assume it is a trace setting
 */

if (options != NULL && strlen(options) > 0) {
    rc = jvmri_intf->TraceSet(env, options);
    if (rc != JNI_OK) {
        fprintf(stderr, "RASplugin004 Return code %d setting trace options\n", rc);
        fflush(stderr);
        return JNI_ERR;
    }
}

```

To set Maximal tracing for 'j9mm', use the following command when launching the JVM and your agent:

```
java -Xrunjvmri:maximal=j9mm -Xtrace:external=j9mm App.class
```

Note: Trace must be enabled before the agent can be used. To do this, specify the trace option on the command-line: `-Xtrace:external=j9mm`.

Starting the agent

To start the agent when the JVM starts up, use the **-Xrun** option. For example if your agent is called `jvmri`, specify `-Xrunjvmri: <options>` on the command-line.

Building the agent

You must set some configuration options before you can build a JVMRI agent.

Building the agent on AIX or Linux

To build a JVMRI agent, write a shell script that contains the following commands:

```
export SDK_BASE=<sdk directory>
export INCLUDE_DIRS="-I. -I$SDK_BASE/include"
export JVM_LIB=-L$SDK_BASE/jre/bin/classic
gcc $INCLUDE_DIRS $JVM_LIB -ljvm -o libmyagent.so -shared myagent.c
```

Where <sdk directory> is the directory where your SDK is installed.

Building the agent on Windows

Before you can build a JVMRI agent, ensure that:

- The agent is contained in a C file called myagent.c.
- You have Microsoft® Visual C/C++ installed.
- The directories sdk\include\ and sdk\include\win32 have been added to the environment variable INCLUDE.

To build a JVMRI agent, enter the command:

```
cl /MD /Femyagent.dll myagent.c /link /DLL
```

Building the agent on z/OS

To build a JVMRI agent, write a shell script that contains the following entries:

```
SDK_BASE= <sdk directory>
USER_DIR= <user agent's source directory>
c++ -c -g -I$SDK_BASE/include -I$USER_DIR -W "c,float(ieee)"
        -W "c,langl1l(extended)" -W "c,expo,dll" myagent.c
c++ -W "l,dll" -o libmyagent.so myagent.o
chmod 755 libmyagent.so
```

This builds a non-xplink library.

Agent design

The agent must reference the header files jni.h and jvmri.h, which are shipped with the SDK and are in the sdk\include subdirectory.

To start the agent, use the -Xrun command-line option. The JVM parses the -Xrunlibrary_name[:options] switch and loads library_name if it exists. A check for an entry point that is called JVM_OnLoad is then made. If the entry point exists, it is called to allow the library to initialize. This processing occurs after the initialization of all JVM subcomponents. The agent can then call the functions that have been initialized, by using the JVMRI table.

JVMRI functions

At startup, the JVM initializes JVMRI. You access the JVMRI functions with the JNI GetEnv() routine to obtain an interface pointer.

For example:

```
JNIEXPORT jint JNICALL
JVM_OnLoad(JavaVM *vm, char *options, void *reserved)
{
    DgRasInterface *ri;
    .....
    (*vm)->GetEnv(vm, (void **)&ri, JVMRAS_VERSION_1_3)
```

```

        rc = jvmas_intf->TraceRegister50(env, listener);
        .....
    }

```

API calls provided by JVMRI

The JVMRI functions are defined in a header file `jvmri.h`, which is supplied in the `sdk/include` directory. Note that all calls must be made using a valid `JNIEnv` pointer as the first parameter.

The `TraceRegister` and `TraceDeregister` functions are deprecated. Use `TraceRegister50` and `TraceDeregister50`.

CreateThread

```

int CreateThread( JNIEnv *env, void JNICALL (*startFunc)(void*),
                 void *args, int GCSuspend)

```

Description

Creates a thread. A thread can be created only after the JVM has been initialized. However, calls to `CreateThread` can be made also before initialization; the threads are created by a callback function after initialization.

Parameters

- A valid pointer to a `JNIEnv`.
- Pointer to start function for the new thread.
- Pointer to argument that is to be passed to start function.
- `GCSuspend` parameter is ignored.

Returns

JNI Return code `JNI_OK` if thread creation is successful; otherwise, `JNI_ERR`.

DumpDeregister

```

int DumpDeregister(JNIEnv *env, int (JNICALL *func)(JNIEnv *env2,
void **threadLocal, int reason))

```

Description

De-registers a dump call back function that was previously registered by a call to `DumpRegister`.

Parameters

- A valid pointer to a `JNIEnv`.
- Function pointer to a previously registered dump function.

Returns

JNI return codes `JNI_OK` and `JNI_EINVAL`.

DumpRegister

```

int DumpRegister(JNIEnv *env, int (JNICALL *func)(JNIEnv *env2,
void **threadLocal, int reason))

```

Description

Registers a function that is called back when the JVM is about to generate a `JavaCore` file.

Parameters

- A valid pointer to a `JNIEnv`.

- Function pointer to dump function to register.

Returns

JNI return codes JNI_OK and JNI_ENOMEM.

DynamicVerbosegc

```
void JNICALL *DynamicVerbosegc (JNIEnv *env, int vgc_switch,
                                int vgccon, char* file_path, int number_of_files,
                                int number_of_cycles);
```

Description

Not supported. Displays the message "not supported".

Parameters

- A valid pointer to a JNIEnv.
- Integer that indicates the direction of switch (JNI_TRUE = on, JNI_FALSE = off)
- Integer that indicates the level of verbosegc (0 = **-verbose:gc**, 1 = **-verbose:Xgccon**)
- Pointer to string that indicates file name for file redirection
- Integer that indicates the number of files for redirection
- Integer that indicates the number of cycles of verbose:gc per file

Returns

None.

GenerateHeapdump

```
int GenerateHeapdump( JNIEnv *env )
```

Description

Generates a Heapdump file.

Parameters

- A valid pointer to a JNIEnv.

Returns

JNI Return code JNI_OK if running dump is successful; otherwise, JNI_ERR.

GenerateJavacore

```
int GenerateJavacore( JNIEnv *env )
```

Description

Generates a Javacore file.

Parameters

- A valid pointer to a JNIEnv.

Returns

JNI Return code JNI_OK if running dump is successful; otherwise, JNI_ERR.

GetComponentDataArea

```
int GetComponentDataArea( JNIEnv *env, char *componentName,
                          void **dataArea, int *dataSize )
```

Description

Not supported. Displays the message "no data area for <requested component>"

Parameters

- A valid pointer to a JNIEnv.

- Component name.
- Pointer to the component data area.
- Size of the data area.

Returns

JNI_ERR

GetRasInfo

```
int GetRasInfo(JNIEnv * env,
               RasInfo * info_ptr)
```

Description

This function fills in the supplied RasInfo structure, based on the request type that is initialized in the RasInfo structure. (See details of the RasInfo structure in “RasInfo structure” on page 382.

Parameters

- A valid pointer to a JNIEnv. This parameter is reserved for future use.
- Pointer to a RasInfo structure. This should have the **type** field initialized to a supported request.

Returns

JNI Return codes JNI_OK, JNI_EINVAL and JNI_ENOMEM.

InitiateSystemDump

```
int JNICALL InitiateSystemDump( JNIEnv *env )
```

Description

Initiates a system dump. The dumps and the output that are produced depend on the settings for JAVA_DUMP_OPTS and JAVA_DUMP_TOOL and on the support that is offered by each platform.

Parameters

- A valid pointer to a JNIEnv.

Returns

JNI Return code JNI_OK if dump initiation is successful; otherwise, JNI_ERR. If a specific platform does not support a system-initiated dump, JNI_EINVAL is returned.

InjectOutOfMemory

```
int InjectOutOfMemory( JNIEnv *env )
```

Description

Causes native memory allocations made after this call to fail. This function is intended to simulate exhaustion of memory allocated by the operating system.

Parameters

- A valid pointer to a JNIEnv.

Returns

JNI_OK if the native allocation function is successfully swapped for the JVMRI function that always returns NULL, JNI_ERR if the swap is unsuccessful.

InjectSigSegv

```
int InjectSigsegv( JNIEnv *env )
```

Description

Raises a SIGSEGV exception, or the equivalent for your platform.

Parameters

- A valid pointer to a JNIEnv.

Returns

JNI_ERR

NotifySignal

```
void NotifySignal(JNIEnv *env, int signal)
```

Description

Raises a signal in the JVM.

Parameters

- A valid pointer to a JNIEnv. This parameter is reserved for future use.
- Signal number to raise.

Returns

Nothing.

ReleaseRasInfo

```
int ReleaseRasInfo(JNIEnv * env,  
RasInfo * info_ptr)
```

Description

This function frees any areas to which the RasInfo structure might point after a successful GetRasInfo call. The request interface never returns pointers to 'live' JVM control blocks or variables.

Parameters

- A valid pointer to a JNIEnv. This parameter is reserved for future use.
- Pointer to a RasInfo structure. This should have previously been set up by a call to GetRasInfo. An error occurs if the **type** field has not been initialized to a supported request. (See details of the RasInfo structure in "RasInfo structure" on page 382.)

Returns

JNI Return codes JNI_OK or JNI_EINVAL.

RunDumpRoutine

```
int RunDumpRoutine( JNIEnv *env, int componentID, int level, void (*printrtn)  
(void *env, const char *tagName, const char *fmt, ...) )
```

Description

Not supported. Displays the message ?not supported?.

Parameters

- A valid pointer to a JNIEnv.
- Id of component to dump.
- Detail level of dump.
- Print routine to which dump output is directed.

Returns

JNI_ERR

SetOutOfMemoryHook

```
int SetOutOfMemoryHook( JNIEnv *env, void (*rasOutOfMemoryHook)
                        (void) )
```

Description

Registers a callback function for an out-of-memory condition.

Parameters

- A valid pointer to a JNIEnv.
- Pointer to callback function.

Returns

JNI Return code JNI_OK if table is successfully updated; otherwise, JNI_ERR.

TraceDeregister

```
int TraceDeregister(JNIEnv *env, void (JNICALL *func)(JNIEnv *env2,
void **threadLocal, int traceId, const char *
format, va_list varargs))
```

Description

Deregisters an external trace listener.

Important: This function is now deprecated. Use “TraceDeregister50.”

Parameters

- A valid pointer to a JNIEnv.
- Function pointer to a previously-registered trace function.

Returns

JNI Return code JNI_OK or JNI_EINVAL.

TraceDeregister50

```
int TraceDeregister50 (
    JNIEnv *env,
    void ( JNICALL *func ) (
        JNIEnv *env2,
        void **threadLocal,
        const char *moduleName,
        int traceId,
        const char *format,
        va_list varargs
    )
)
```

Description

Deregisters an external trace listener.

Parameters

- A valid pointer to a JNIEnv.
- Function pointer to a previously-registered trace function.

Returns

JNI Return code JNI_OK or JNI_EINVAL.

TraceRegister

```
int TraceRegister(JNIEnv *env, void (JNICALL *func)(JNIEnv *env2,
void **threadLocal, int traceId, const char * format,
va_list var))
```

Description

Registers a trace listener.

Important: This function is now deprecated. Use “TraceRegister50.”

Parameters

- A valid pointer to a JNIEnv.
- Function pointer to trace function to register.

Returns

JNI Return code JNI_OK or JNI_ENOMEM.

TraceRegister50

```
int TraceRegister50 (
    JNIEnv *env,
    void ( JNICALL *func ) (
        JNIEnv *env2,
        void **threadLocal,
        const char *moduleName,
        int traceId,
        const char *format,
        va_list varargs
    )
)
```

Description

Registers a trace listener.

Parameters

- A valid pointer to a JNIEnv.
- Function pointer to trace function to register.

Returns

JNI Return code JNI_OK or JNI_ENOMEM.

TraceResume

```
void TraceResume(JNIEnv *env)
```

Description

Resumes tracing.

Parameters

- A valid pointer to a JNIEnv. If MULTI_JVM; otherwise, it can be NULL.

Returns

Nothing.

TraceResumeThis

```
void TraceResumeThis(JNIEnv *env);
```

Description

Resume tracing from the current thread. This action decrements the *resumeCount* for this thread. When it reaches zero (or below) the thread *starts* tracing (see Chapter 25, “Tracing Java applications and the JVM,” on page 283).

Parameters

- A valid pointer to a JNIEnv.

Returns
None.

TraceSet

```
int TraceSet(JNIEnv *env, const char *cmd)
```

Description

Sets the trace configuration options. This call parses only the first valid trace command passed to it, but can be called multiple times. Hence, to achieve the equivalent of setting `-Xtrace:maximal=j9mm,iprint=j9shr`, you call `TraceSet` twice, once with the `cmd` parameter `maximal=j9mm` and once with `iprint=j9shr`.

Parameters

- A valid pointer to a `JNIEnv`.
- Trace configuration command.

Returns

JNI Return code `JNI_OK`, `JNI_ERR`, `JNI_ENOMEM`, `JNI_EXIST` and `JNI_EINVAL`.

TraceSnap

```
void TraceSnap(JNIEnv *env, char *buffer)
```

Description

Takes a snapshot of the current trace buffers.

Parameters

- A valid pointer to a `JNIEnv`; if set to `NULL`, current `Execenv` is used.
- The second parameter is no longer used, but still exists to prevent changing the function interface. It can safely be set to `NULL`.

Returns

Nothing

TraceSuspend

```
void TraceSuspend(JNIEnv *env)
```

Description

Suspends tracing.

Parameters

- A valid pointer to a `JNIEnv`; if `MULTI_JVM`; otherwise, it can be `NULL`.

Returns

Nothing.

TraceSuspendThis

```
void TraceSuspendThis(JNIEnv *env);
```

Description

Suspend tracing from the current thread. This action decrements the *suspendcount* for this thread. When it reaches zero (or below) the thread *stops* tracing (see Chapter 25, “Tracing Java applications and the JVM,” on page 283).

Parameters

- A valid pointer to a `JNIEnv`.

Returns

None.

RasInfo structure

The RasInfo structure that is used by GetRasInfo() takes the following form. (Fields that are initialized by GetRasInfo are underscored):

```
typedef struct RasInfo {
    int type;
    union {
        struct {
            int number;
            char **names;
        } query;
        struct {
            int number;
            char **names;
        } trace_components;
        struct {
            char *name
            int first;
            int last;
            unsigned char *bitMap;
        } trace_component;
        } info;
    } RasInfo;
```

RasInfo request types

The following request types are supported:

RASINFO_TYPES

Returns the *number* of request types that are supported and an array of pointers to their names in the enumerated sequence. The names are in code page ISO8859-1.

RASINFO_TRACE_COMPONENTS

Returns the *number* of components that can be enabled for trace and an array of pointers to their *names* in the enumerated sequence. The names are in code page ISO8859-1.

RASINFO_TRACE_COMPONENT

Returns the *first* and *last* tracepoint ids for the component *name* (code page ISO8859-1) and a *bitmap* of those tracepoints, where a 1 signifies that the tracepoint is in the build. The *bitmap* is big endian (tracepoint id *first* is the most significant bit in the first byte) and is of length ((last-first)+7)/8 bytes.

Intercepting trace data

To receive trace information from the JVM, you can register a trace listener using JVMRI. In addition, you must specify the option **-Xtrace:external=<option>** to route trace information to an external trace listener.

The -Xtrace:external=<option>

The format of this property is:

```
-Xtrace:external=[(!)]tracepoint_specification[,...]
```

This system property controls what is traced. Multiple statements are allowed and their effect is cumulative.

The *tracepoint_specification* is as follows:

Component[(*Class*[,...])]

Where *component* is the JVM subcomponent or **all**. If no component is specified, **all** is assumed.

class is the tracepoint type or **all**. If class is not specified, **all** is assumed.

TPID(*tracepoint_id*[,...])

Where *tracepoint_id* is the hexadecimal global tracepoint identifier.

If no qualifier parameters are entered, all tracepoints are enabled; that is, the equivalent of specifying **all**.

The ! (exclamation mark) is a logical not. It allows complex tracepoint selection.

Calling external trace

If an external trace routine has been registered and a tracepoint has been enabled for external trace, it is called with the following parameters:

env

Pointer to the JNIEnv for the current thread.

traceid

Trace identifier

format

A zero-terminated string that describes the format of the variable argument list that follows. The possible values for each character position are:

0x01	One character
0x02	Short
0x04	Int
0x08	Double or long long
0xfe	Pointer to java/lang/String object
0xff	ASCII string pointer (can be NULL)
0x00	End of format string

If the format pointer is NULL, no trace data follows.

varargs

A *va_list* of zero or more arguments as defined in **format** argument.

Formatting

You can use J9TraceFormat.dat to format JVM-generated tracepoints that are captured by the agent. J9TraceFormat.dat is shipped with the SDK.

J9TraceFormat.dat consists of a flat ASCII or EBCDIC file of the following format:

5.0

j9vm 0 1 1 N Trc_VM_VMInitStages_Event1 " Trace engine initialized for module j9vm"

j9vm 2 1 1 N Trc_VM_CreateRAMClassFromROMClass_Entry " >Create RAM class from ROM class %p in clas

j9vm 4 1 1 N Trc_VM_CreateRAMClassFromROMClass_Exit " j9vm 4 1 1 N Trc_VM_CreateRAMClassFromROMCla

The first line contains the version number of the format file. A new version number reflects changes to the layout of this file.

The format of each tracepoint entry is as follows:

<component> <t> <o> <l> <e> <symbol> <template>

where:

- *<component>* is the internal JVM component name.
- *<t>* is the tracepoint type (0 through 11).
- *<o>* is the overhead (0 through 10).
- *<l>* is the level of the tracepoint (0 through 9, or - if the tracepoint is obsolete).
- *<e>* is the explicit setting flag (Y/N).
- *<symbol>* is the name of the tracepoint.
- *<template>* is a template that is used to format the entry. The template consists of the text that appears in double quotation marks (").

Tracepoint types are as follows:

Type 0

Event

Type 1

Exception

Type 2

Entry

Type 4

Exit

Type 5

Exit-with-Exception

Type 6

Mem

Any other type is reserved for development use; you should not find any other type on a release version of IBM Java.

Note: This condition is subject to change without notice.

The version number is different for each version.

Chapter 32. Using the HPROF Profiler

HPROF is a demonstration profiler shipped with the IBM SDK that uses the JVMTI to collect and record information about Java execution. Use it to work out which parts of a program are using the most memory or processor time.

To improve the efficiency of your applications, you must know which parts of the code are using large amounts of memory and processor resources. HPROF is an example JVMTI agent and is started using the following syntax:

```
java -Xrunhprof[:<option>=<value>,...] <classname>
```

When you run Java with HPROF, a file is created when the program ends. This file is placed in the current working directory and is called `java.hprof.txt` (`java.hprof` if binary format is used) unless a different file name has been given. This file contains a number of different sections, but the exact format and content depend on the selected options.

If you need more information about HPROF than is contained in this section, see <http://java.sun.com/developer/technicalArticles/Programming/HPROF.html>.

The command `java -Xrunhprof:help` shows the options available:

heap=dump | sites | all

This option helps in the analysis of memory usage. It tells HPROF to generate stack traces, from which you can see where memory was allocated. If you use the **heap=dump** option, you get a dump of all live objects in the heap. With **heap=sites**, you get a sorted list of sites with the most heavily allocated objects at the top. The default value **all** gives both types of output.

cpu=samples | times | old

The **cpu** option provides information that is useful in determining where the processor spends most of its time. If **cpu** is set to **samples**, the JVM pauses execution and identifies which method call is active. If the sampling rate is high enough, you get a good picture of where your program spends most of its time. If **cpu** is set to **time**, you receive precise measurements of how many times each method was called and how long each execution took. Although this option is more accurate, it slows down the program. If **cpu** is set to **old**, the profiling data is produced in the old HPROF format.

interval=y | n

The interval option applies only to **cpu=samples** and controls the time that the sampling thread sleeps between samples of the thread stacks.

monitor=y | n

The **monitor** option can help you understand how synchronization affects the performance of your application. Monitors implement thread synchronization. Getting information about monitors can tell you how much time different threads are spending when trying to access resources that are already locked. HPROF also gives you a snapshot of the monitors in use. This information is useful for detecting deadlocks.

format=a | b

The default for the output file is ASCII format. Set **format** to 'b' if you want to specify a binary format, which is required for some utilities like the Heap Analysis Tool.

file=<filename>

Use the **file** option to change the name of the output file. The default name for an ASCII file is `java.hprof.txt`. The default name for a binary file is `java.hprof`.

force=y | n

Typically, the default (**force=y**) overwrites any existing information in the output file. So, if you have multiple JVMs running with HPROF enabled, use **force=n**, which appends additional characters to the output file name as needed.

net=<host>:<port>

To send the output over the network rather than to a local file, use the **net** option.

depth=<size>

The **depth** option indicates the number of method frames to display in a stack trace. The default is 4.

thread=y | n

If you set the **thread** option to **y**, the thread id is printed beside each trace. This option is useful if you cannot see which thread is associated with which trace. This type of problem might occur in a multi-threaded application.

doe=y | n

The default behavior is to collect profile information when an application exits. To collect the profiling data during execution, set **doe** (dump on exit) to **n**.

msa=y | n

The **msa** option applies only to Solaris and causes the Solaris Micro State Accounting to be used. This feature is unsupported on IBM SDK platforms.

cutoff=<value>

Many sample entries are produced for a small percentage of the total execution time. By default, HPROF includes all execution paths that represent at least 0.0001 percent of the time spent by the processor. You can increase or decrease that cutoff point using this option. For example, to eliminate all entries that represent less than one-fourth of one percent of the total execution time, you specify **cutoff=0.0025**.

verbose=y | n

This option generates a message when dumps are taken. The default is **y**.

lineno=y | n

Each frame typically includes the line number that was processed, but you can use this option to suppress the line numbers from the output listing. If enabled, each frame contains the text `Unknown line` instead of the line number.

```
TRACE 1056:
java/util/Locale.toUpperCase(Locale.java:Unknown line)
java/util/Locale.<init>(Locale.java:Unknown line)
java/util/Locale.<clinit>(Locale.java:Unknown line)
sun/io/CharacterEncoding.aliasName(CharacterEncoding.java:Unknown line)
```

Explanation of the HPROF output file

The top of the file contains general header information such as an explanation of the options, copyright, and disclaimers. A summary of each thread follows.

You can see the output after using HPROF with a simple program, shown as follows. This test program creates and runs two threads for a short time. From the output, you can see that the two threads called `apples` and then `oranges` were

created after the system-generated main thread. Both threads end before the main thread. For each thread its address, identifier, name, and thread group name are displayed. You can see the order in which threads start and finish.

```
THREAD START (obj=11199050, id = 1, name="Signal dispatcher", group="system")
THREAD START (obj=111a2120, id = 2, name="Reference Handler", group="system")
THREAD START (obj=111ad910, id = 3, name="Finalizer", group="system")
THREAD START (obj=8b87a0, id = 4, name="main", group="main")
THREAD END (id = 4)
THREAD START (obj=11262d18, id = 5, name="Thread-0", group="main")
THREAD START (obj=112e9250, id = 6, name="apples", group="main")
THREAD START (obj=112e9998, id = 7, name="oranges", group="main")
THREAD END (id = 6)
THREAD END (id = 7)
THREAD END (id = 5)
```

The trace output section contains regular stack trace information. The depth of each trace can be set and each trace has a unique ID:

```
TRACE 5:
  java/util/Locale.toLowerCase(Locale.java:1188)
  java/util/Locale.convertOldISOCodes(Locale.java:1226)
  java/util/Locale.<init>(Locale.java:273)
  java/util/Locale.<clinit>(Locale.java:200)
```

A trace contains a number of frames, and each frame contains the class name, method name, file name, and line number. In the previous example, you can see that line number 1188 of `Locale.java` (which is in the `toLowerCase` method) has been called from the `convertOldISOCodes()` function in the same class. These traces are useful in following the execution path of your program. If you set the monitor option, a monitor dump is produced that looks like this example:

```
MONITOR DUMP BEGIN
  THREAD 8, trace 1, status: R
  THREAD 4, trace 5, status: CW
  THREAD 2, trace 6, status: CW
  THREAD 1, trace 1, status: R
  MONITOR java/lang/ref/Reference$Lock(811bd50) unowned
  waiting to be notified: thread 2
  MONITOR java/lang/ref/ReferenceQueue$Lock(8134710) unowned
  waiting to be notified: thread 4
  RAW MONITOR "_hprof_dump_lock"(0x806d7d0)
  owner: thread 8, entry count: 1
  RAW MONITOR "Monitor Cache lock"(0x8058c50)
  owner: thread 8, entry count: 1
  RAW MONITOR "Monitor Registry lock"(0x8058d10)
  owner: thread 8, entry count: 1
  RAW MONITOR "Thread queue lock"(0x8058bc8)
  owner: thread 8, entry count: 1
MONITOR DUMP END
MONITOR TIME BEGIN (total = 0 ms) Thu Aug 29 16:41:59 2002
MONITOR TIME END
```

The first part of the monitor dump contains a list of threads, including the trace entry that identifies the code the thread executed. There is also a thread status for each thread where:

- R — Runnable (The thread is able to run when given the chance)
- S — Suspended (The thread has been suspended by another thread)
- CW — Condition Wait (The thread is waiting)
- MW — Monitor Wait (The monitor is waiting)

Next is a list of monitors along with their owners and an indication of whether there are any threads waiting on them.

The Heapdump is the next section. This information contains a list of the different areas of memory, and shows how they are allocated:

```
CLS 1123edb0 (name=java/lang/StringBuffer, trace=1318)
  super 111504e8
  constant[25] 8abd48
  constant[32] 1123edb0
  constant[33] 111504e8
  constant[34] 8aad38
  constant[115] 1118cdc8
CLS 111ecff8 (name=java/util/Locale, trace=1130)
  super 111504e8
  constant[2] 1117a5b0
  constant[17] 1124d600
  constant[24] 111fc338
  constant[26] 8abd48
  constant[30] 111fc2d0
  constant[34] 111fc3a0
  constant[59] 111ecff8
  constant[74] 111504e8
  constant[102] 1124d668
  ...
CLS 111504e8 (name=java/lang/Object, trace=1)
  constant[18] 111504e8
```

CLS tells you that memory is being allocated for a class. The hexadecimal number following it is the address where that memory is allocated.

Next is the class name followed by a trace reference. Use this information to cross-reference the trace output and see when the class is called. If you refer to that particular trace, you can get the line number of the instruction that led to the creation of this object. The addresses of the constants in this class are also displayed and, in the previous example, the address of the class definition for the superclass. Both classes are a child of the same superclass (with address 11504e8). Looking further through the output, you can see this class definition and name. It is the Object class (a class that every class inherits from). The JVM loads the entire superclass hierarchy before it can use a subclass. Thus, class definitions for all superclasses are always present. There are also entries for Objects (OBJ) and Arrays (ARR):

```
OBJ 111a9e78 (sz=60, trace=1, class=java/lang/Thread@8b0c38)
  name 111afbfb8
  group 111af978
  contextClassLoader 1128fa50
  inheritedAccessControlContext 111aa2f0
  threadLocals 111bea08
  inheritableThreadLocals 111bea08
ARR 8bb978 (sz=4, trace=2, nelems=0, elem type=java/io/ObjectStreamField@8bac80)
```

If you set the **heap** option to **sites** or **all**, you get a list of each area of storage allocated by your code. The parameter **all** combines **dump** and **sites**. This list is ordered with the sites that allocate the most memory at the top:

```
SITES BEGIN (ordered by live bytes) Tue Feb 06 10:54:46 2007
```

	percent		live		alloc'ed		stack	class
rank	self	accum	bytes	objs	bytes	objs	trace	name
1	20.36%	20.36%	190060	16	190060	16	300000	byte[]
2	14.92%	35.28%	139260	1059	139260	1059	300000	char[]
3	5.27%	40.56%	49192	15	49192	15	300055	byte[]
4	5.26%	45.82%	49112	14	49112	14	300066	byte[]
5	4.32%	50.14%	40308	1226	40308	1226	300000	java.lang.String
6	1.62%	51.75%	15092	438	15092	438	300000	java.util.HashMap\$Entry
7	0.79%	52.55%	7392	14	7392	14	300065	byte[]
8	0.47%	53.01%	4360	16	4360	16	300016	char[]
9	0.47%	53.48%	4352	34	4352	34	300032	char[]

10	0.43%	53.90%	3968	32	3968	32	300028	char[]
11	0.40%	54.30%	3716	8	3716	8	300000	java.util.HashMap\$Entry[]
12	0.40%	54.70%	3708	11	3708	11	300000	int[]
13	0.31%	55.01%	2860	16	2860	16	300000	java.lang.Object[]
14	0.28%	55.29%	2644	65	2644	65	300000	java.util.Hashtable\$Entry
15	0.28%	55.57%	2640	15	2640	15	300069	char[]
16	0.27%	55.84%	2476	17	2476	17	300000	java.util.Hashtable\$Entry[]
17	0.25%	56.08%	2312	16	2312	16	300013	char[]
18	0.25%	56.33%	2312	16	2312	16	300015	char[]
19	0.24%	56.57%	2224	10	2224	10	300000	java.lang.Class

In this example, Trace 300055 allocated 5.27% of the total allocated memory. This percentage works out to be 49192 bytes.

The **cpu** option gives profiling information about the processor. If **cpu** is set to samples, the output contains the results of periodic samples taken during execution of the code. At each sample, the code path being processed is recorded, and a report is produced similar to:

```
CPU SAMPLES BEGIN (total = 714) Fri Aug 30 15:37:16 2002
rank  self  accum   count trace method
1 76.28% 76.28% 501 77 MyThread2.bigMethod
2  6.92% 83.20%  47 75 MyThread2.smallMethod
...
CPU SAMPLES END
```

You can see that the bigMethod() was responsible for 76.28% of the processor execution time and was being run 501 times out of the 714 samples. If you use the trace IDs, you can see the exact route that led to this method being called.

Chapter 33. Using the JVMTI

JVMTI is a two-way interface that allows communication between the JVM and a native agent. It replaces the JVMDI and JVMPI interfaces.

JVMTI allows third parties to develop debugging, profiling, and monitoring tools for the JVM. The interface contains mechanisms for the agent to notify the JVM about the kinds of information it requires. The interface also provides a means of receiving the relevant notifications. Several agents can be attached to a JVM at any one time. A number of tools are based on this interface, such as Hyades, JProfiler, and Ariadna. These are third-party tools, therefore IBM cannot make any guarantees or recommendations regarding them. IBM does provide a simple profiling agent based on this interface, HPROF. For details about its use, see Chapter 32, “Using the HPROF Profiler,” on page 385.

JVMTI agents can be loaded at startup using short or long forms of the command-line option:

`-agentlib:<agent-lib-name>=<options>`

or

`-agentpath:<path-to-agent>=<options>`

For example:

`-agentlib:hprof=<options>`

assumes that a folder containing `hprof.dll` is on the library path, or

`-agentpath:C:\sdk\jre\bin\hprof.dll=<options>`

For more information about JVMTI, see <http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html>.

For advice on porting JVMPI-based profilers to JVMTI, see <http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition>.

For a guide about writing a JVMTI agent, see <http://java.sun.com/developer/technicalArticles/Programming/jvmti>.

Attention: When the class redefinition capability is requested, the JVM for IBM Java 5.0 operates under Full Speed Debug (FSD) mode. This mode can have a significant impact on application performance.

For more information about class redefinition, see <http://java.sun.com/j2se/1.5.0/docs/api/java/lang/instrument/package-summary.html>.

Chapter 34. Using the Diagnostic Tool Framework for Java

The Diagnostic Tool Framework for Java (DTFJ) is a Java application programming interface (API) from IBM used to support the building of Java diagnostics tools. DTFJ works with data from a system dump.

To work with a system dump, the dump must be processed by the jextract tool; see “Using the dump extractor, jextract” on page 264. The jextract tool produces metadata from the dump, which allows the internal structure of the JVM to be analyzed. You must run jextract on the system that produced the dump.

The DTFJ API helps diagnostics tools access the following information:

- Memory locations stored in the dump
- Relationships between memory locations and Java internals
- Java threads running in the JVM
- Native threads held in the dump
- Java classes and their classloaders that were present
- Java objects that were present in the heap
- Java monitors and the objects and threads they are associated with
- Details of the workstation on which the dump was produced
- Details of the Java version that was being used
- The command line that launched the JVM

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 397

The full details of the DTFJ Interface are provided with the SDK as Javadoc information in `sdk/docs/apidoc.zip`. DTFJ classes are accessible without modification to the class path.

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 5.0.

All DTFJ implementations support the same interface, but the DTFJ implementation supplied in Version 5.0 is different to the implementation supplied in Version 1.4.2. The DTFJ implementations have different factory class names that you must use. The DTFJ implementation supplied in Version 1.4.2 does not work with system dumps from Version 5.0, and the DTFJ implementation supplied in Versions 5.0 does not work with system dumps from Version 1.4.2.

Figure 2 on page 396 illustrates the DTFJ interface. The starting point for working with a dump is to obtain an Image instance by using the ImageFactory class supplied with the concrete implementation of the API.

Working with a system dump

The following example shows how to work with a system dump.

```
import java.io.File;
import java.util.Iterator;
import java.io.IOException;

import com.ibm.dtfj.image.CorruptData;
import com.ibm.dtfj.image.Image;
import com.ibm.dtfj.image.ImageFactory;

public class DTFJEX1 {
    public static void main(String[] args) {
        Image image = null;
        if (args.length > 0) {
            File f = new File(args[0]);
            try {
                Class factoryClass = Class
                    .forName("com.ibm.dtfj.image.j9.ImageFactory");
                ImageFactory factory = (ImageFactory) factoryClass
                    .newInstance();
                image = factory.getImage(f);
            } catch (ClassNotFoundException e) {
                System.err.println("Could not find DTFJ factory class");
                e.printStackTrace(System.err);
            } catch (IllegalAccessException e) {
                System.err.println("IllegalAccessException for DTFJ factory class");
                e.printStackTrace(System.err);
            } catch (InstantiationException e) {
                System.err.println("Could not instantiate DTFJ factory class");
                e.printStackTrace(System.err);
            } catch (IOException e) {
                System.err.println("Could not find/use required file(s)");
                e.printStackTrace(System.err);
            }
        } else {
            System.err.println("No filename specified");
        }
        if (image == null) {
            return;
        }

        Iterator asIt = image.getAddressSpaces();
        int count = 0;
        while (asIt.hasNext()) {
            Object tempObj = asIt.next();
            if (tempObj instanceof CorruptData) {
                System.err.println("Address Space object is corrupt: "
                    + (CorruptData) tempObj);
            } else {
                count++;
            }
        }
        System.out.println("The number of address spaces is: " + count);
    }
}
```

In this example, the only section of code that ties the dump to a particular implementation of DTFJ is the generation of the factory class. Change the factory to use a different implementation.

The `getImage()` methods in `ImageFactory` expect one file, the `dumpfilename.zip` file produced by `jextract` (see “Using the dump extractor, `jextract`” on page 264). If the `getImage()` methods are called with two files, they are interpreted as the dump itself and the `.xml` metadata file. If there is a problem with the file specified, an `IOException` is thrown by `getImage()` and can be caught and (in the example above) an appropriate message issued. If a missing file was passed to the above example, the following output is produced:

```
Could not find/use required file(s)
java.io.FileNotFoundException: core_file.xml (The system cannot find the file specified.)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:135)
  at com.ibm.dtfj.image.j9.ImageFactory.getImage(ImageFactory.java:47)
  at com.ibm.dtfj.image.j9.ImageFactory.getImage(ImageFactory.java:35)
  at DTFJEX1.main(DTFJEX1.java:23)
```

In the case above, the DTFJ implementation is expecting a dump file to exist. Different errors are caught if the file existed but was not recognized as a valid dump file.

After you have obtained an `Image` instance, you can begin analyzing the dump. The `Image` instance is the second instance in the class hierarchy for DTFJ illustrated by the following diagram:

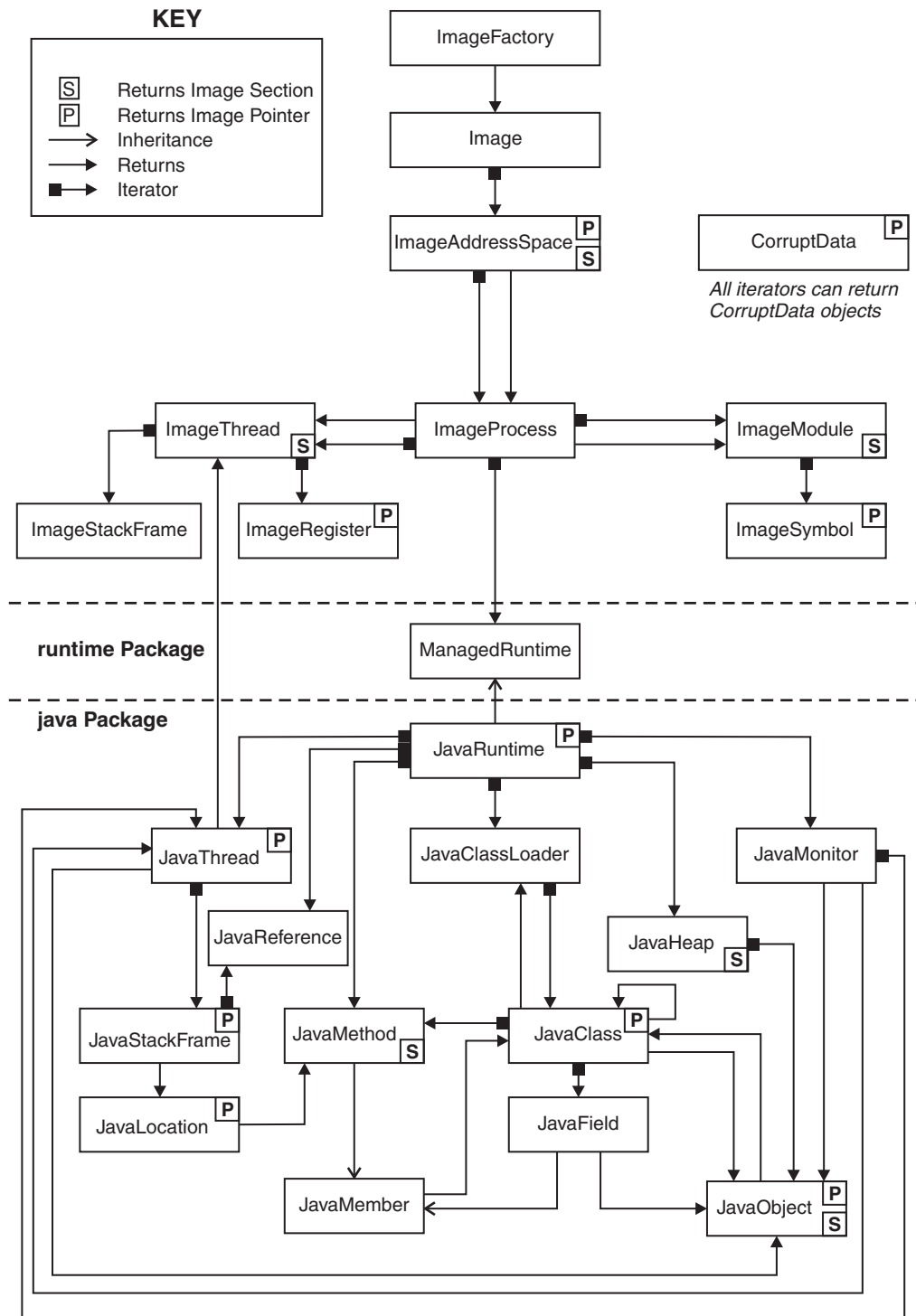


Figure 2. DTFJ interface diagram

The hierarchy displays some major points of DTFJ. Firstly, there is a separation between the Image (the dump, a sequence of bytes with different contents on different platforms) and the Java internal knowledge.

Some things to note from the diagram:

- The DTFJ interface is separated into two parts: classes with names that start with *Image* and classes with names that start with *Java*.
- *Image* and *Java* classes are linked using a *ManagedRuntime* (which is extended by *JavaRuntime*).
- An *Image* object contains one *ImageAddressSpace* object (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*, although for the purposes of this document discussion is limited to the IBM JVM implementations.
- There is a link from a *JavaThread* object to its corresponding *ImageThread* object. Use this link to find out about native code associated with a Java thread, for example JNI functions that have been called from Java.
- If a *JavaThread* was not running Java code when the dump was taken, the *JavaThread* object will have no *JavaStackFrame* objects. In these cases, use the link to the corresponding *ImageThread* object to find out what native code was running in that thread. This is typically the case with the JIT compilation thread and Garbage Collection threads.

DTFJ example application

This example is a fully working DTFJ application.

For clarity, this example does not perform full error checking when constructing the main *Image* object and does not perform *CorruptData* handling in all of the iterators. In a production environment, you use the techniques illustrated in the example in the “Using the DTFJ interface” on page 393.

In this example, the program iterates through every available Java thread and checks whether it is equal to any of the available image threads. When they are found to be equal, the program declares that it has, in this case, “Found a match”.

The example demonstrates:

- How to iterate down through the class hierarchy.
- How to handle *CorruptData* objects from the iterators.
- The use of the *.equals* method for testing equality between objects.

```
import java.io.File;
import java.util.Iterator;
import com.ibm.dtfj.image.CorruptData;
import com.ibm.dtfj.image.CorruptDataException;
import com.ibm.dtfj.image.DataUnavailable;
import com.ibm.dtfj.image.Image;
import com.ibm.dtfj.image.ImageAddressSpace;
import com.ibm.dtfj.image.ImageFactory;
import com.ibm.dtfj.image.ImageProcess;
import com.ibm.dtfj.java.JavaRuntime;
import com.ibm.dtfj.java.JavaThread;
import com.ibm.dtfj.image.ImageThread;

public class DTFJEX2
{
    public static void main( String[] args )
    {
        Image image = null;
        if ( args.length > 0 )
        {
```

```

File f = new File( args[0] );
try
{
    Class factoryClass = Class
        .forName( "com.ibm.dtfj.image.j9.ImageFactory" );
    ImageFactory factory = (ImageFactory) factoryClass.newInstance( );
    image = factory.getImage( f );
}
catch ( Exception ex )
{ /*
    * Should use the error handling as shown in DTFJEX1.
    */
    System.err.println( "Error in DTFJEX2" );
    ex.printStackTrace( System.err );
}
}
else
{
    System.err.println( "No filename specified" );
}

if ( null == image )
{
    return;
}

MatchingThreads( image );
}

public static void MatchingThreads( Image image )
{
    ImageThread imgThread = null;

    Iterator asIt = image.getAddressSpaces( );
    while ( asIt.hasNext( ) )
    {
        System.out.println( "Found ImageAddressSpace..." );

        ImageAddressSpace as = (ImageAddressSpace) asIt.next( );

        Iterator prIt = as.getProcesses( );

        while ( prIt.hasNext( ) )
        {
            System.out.println( "Found ImageProcess..." );

            ImageProcess process = (ImageProcess) prIt.next( );

            Iterator runTimesIt = process.getRuntimes( );
            while ( runTimesIt.hasNext( ) )
            {
                System.out.println( "Found Runtime..." );
                JavaRuntime javaRT = (JavaRuntime) runTimesIt.next( );

                Iterator javaThreadIt = javaRT.getThreads( );

                while ( javaThreadIt.hasNext( ) )
                {
                    Object tempObj = javaThreadIt.next( );
                    /*
                     * Should use CorruptData handling for all iterators
                     */
                    if ( tempObj instanceof CorruptData )
                    {
                        System.out.println( "We have some corrupt data" );
                    }
                    else

```

Chapter 35. Using JConsole

JConsole (Java Monitoring and Management Console) is a graphical tool which allows the user to monitor and manage the behavior of Java applications.

The tool is built on top of the `java.lang.management` API which was introduced in Java 5.0. JConsole connects to applications running on the same workstation as itself, or on a remote workstation. The applications must be configured to allow access. JConsole is not part of the core SDK, and it is experimental and unsupported.

When JConsole connects to a Java application, it reports information about the application. The details include memory usage, the running threads, and the loaded classes. This data allows you to monitor the behavior of your application and the JVM. The information is useful in understanding performance problems, memory usage issues, hangs, or deadlocks.

Setting up JConsole to monitor a Java application

1. The Java application you want to monitor must be started with command-line options which make it accessible to JConsole. The simplest set of options for monitoring are:

```
-Dcom.sun.management.jmxremote.port=<port number>  
-Dcom.sun.management.jmxremote.authenticate=false  
-Dcom.sun.management.jmxremote.ssl=false
```

<port number> is a free port on your workstation. In this example, the `authenticate` and `ssl` options prevent password authentication and encryption using Secure Sockets Layer (SSL). Using these options allow JConsole, or any other JMX agent, to connect to your Java application if it has access to the specified port. Only use these non-secure options in a development or testing environment. For more information about configuring security options, see <http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html#remote>.

2. Start JConsole by typing `jconsole` at a command prompt. Your path must contain the `bin` directory of the SDK.
3. The JConsole **Connect to Agent** dialog opens: Enter the port number that you specified in step 1. If you are running JConsole on the same workstation as your Java application, leave the host name value as `localhost`. For a remote system, set the host field value to the host name or IP address of the workstation. Leave the **Username** and **Password** fields blank if you used the options specified in step 1.
4. Click **connect**. JConsole starts and displays the summary tab.

Setting up JConsole to monitor itself

JConsole can monitor itself. This ability is useful for simple troubleshooting of the Java environment.

1. Start JConsole by typing `jconsole` at a command prompt. Your path must contain the `bin` directory of the SDK.
2. The JConsole **Connect to Agent** dialog opens: Enter `localhost` in the host field, and `0` in the port field.
3. Click **connect**. JConsole starts and displays the summary tab.

Using JConsole to monitor a Java application

The JConsole summary tab shows key details of the JVM you have connected to. From here, you can select any of the other tabs for more details on a particular aspect. The Memory tab shows a history of usage of each memory pool in the JVM, – the most useful being the heap memory usage.

You can also request that a GC is carried out by clicking the **Perform GC** button. You must be connected with security options disabled as described previously, or be authenticated as a control user.

The Threads tab shows the number of threads currently running and a list of their IDs.

Clicking a thread ID shows the thread state and its current stack trace.

The Classes tab displays the current number of loaded classes and the number of classes loaded and unloaded since the application was started. Selecting the **verbose output** check box allows verbose class loading output to be switched on and off to see a list of classes that are loaded in the client JVM. The output is displayed on the stderr output of the client JVM.

The MBeans tab allows you to inspect the state of the platform MBeans, which provides more detail about the JVM.

Clicking an MBean in the MBean tab provides a set of further tabs relating to that particular MBean; Attributes, Operations, Notifications, and Info. “Attributes” provide information about the current state of the JVM. Some attributes allow you to change the state of the JVM. For example, inside the **Memory** tab, enabling the **Verbose** option turns on VerboseGC logging. “Operations” allow you to get more in-depth information back from the JVM. For example, inside the **Threading** tab you see thread information. You can use this information to identify any monitor-deadlocked threads. Some MBeans provide notifications that JConsole is able to subscribe to. These notifications are accessed in the Notifications tab. The notifications available are documented in the Info tab.

See the API documentation for the `java.lang.management` package at <http://java.sun.com/j2se/1.5.0/docs/api/java/lang/management/package-summary.html> for more details on the individual platform MBeans.

Finally, the VM tab gives information about the environment in which your Java application is running including any JVM arguments and the current class path.

Troubleshooting JConsole

JConsole is a Swing application. You might find that running JConsole on the same workstation as the Java application you want to monitor affects the performance of your Java application. You can use JConsole to connect to a JVM running on a remote workstation to reduce the affect of running JConsole on the application performance.

Because JConsole is a Java application, you can pass it Java command-line options through the application that starts JConsole by prefixing them with **-J**. For example, to change the maximum heap size that JConsole uses, add the command-line option **-J-Xmx<size>**.

Further information

More details about JConsole and the definitions of the values it displays can be found at <http://java.sun.com/j2se/1.5.0/docs/guide/management/>.

Chapter 36. Using the IBM Monitoring and Diagnostic Tools for Java - Health Center

The IBM Monitoring and Diagnostic Tools for Java - Health Center is a diagnostic tool for monitoring the status of a running Java Virtual Machine (JVM).

| Information about the IBM Monitoring and Diagnostic Tools for Java - Health
| Center is available on developerWorks and in an InfoCenter.

Part 5. Appendixes

Appendix A. 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 196 for more information about minor codes.

When an error occurs, you might find additional details in the ORB FFDC log. By default, the Java ORB from IBM creates an FFDC log with a filename in the format of `orbtrc.DDMMYYY.HHmm.SS.txt`. If the ORB is operating in the WebSphere Application Server or other IBM product, see the publications for that product to determine the location of the FFDC log.

CONN_CLOSE_REBIND CONN_CLOSE_REBIND

Explanation: An attempt has been made to write to a TCP/IP connection that is closing.

System action: `org.omg.CORBA.COMM_FAILURE`

User response: Ensure that the completion status that is associated with the minor code is **NO**, then reissue the request.

CONN_PURGE_ABORT CONN_PURGE_ABORT

Explanation: An unrecoverable error occurred on a TCP/IP connection. All outstanding requests are cancelled. Errors include:

- A GIOP MessageError or unknown message type
- An IOException that is received while data is being read from the socket
- An unexpected error or exception that occurs during message processing

System action: `org.omg.CORBA.COMM_FAILURE`

User response: Investigate each request and reissue if necessary. If the problem occurs again, enable ORB, network tracing, or both, to determine the cause of the failure.

CONNECT_FAILURE_1 CONNECT_FAILURE_1

Explanation: The client attempted to open a connection with the server, but failed. The reasons for the failure can be many; for example, the server might not be up or it might not be listening on that port. If a BindException is caught, it shows that the client could not open a socket locally (that is, the local port was in use or the client has no local address).

System action: `org.omg.CORBA.TRANIENT`

User response: As with all TRANIENT exceptions, a retry or restart of the client or server might solve the problem. Ensure that the port and server host names

are correct, and that the server is running and allowing connections. Also ensure that no firewall is blocking the connection, and that a route is available between client and server.

CONNECT_FAILURE_5 CONNECT_FAILURE_5

Explanation: An attempt to connect to a server failed with both the direct and indirect IORs. Every client side handle to a server object (managed by the ClientDelegate reference) is set up with two IORs (object references) to reach the servant on the server. The first IOR is the direct IOR, which holds details of the server hosting the object. The second IOR is the indirect IOR, which holds a reference to a naming server that can be queried if the direct IOR “does not work”.

Note: The two IORs might be the same at times. For any remote request, the ORB tries to reach the servant object using the direct IOR and then the indirect IOR. The CONNECT_FAILURE_5 exception is thrown when the ORB failed with both IORs.

System action: `org.omg.CORBA.TRANIENT` (minor code E07)

User response: The cause of failure is typically connection-related, for example because of “connection refused” exceptions. Other CORBA exceptions such as NO_IMPLEMENT or OBJECT_NOT_EXIST might also be the root cause of the (E07) CORBA.TRANIENT exception. An abstract of the root exception is logged in the description of the (E07) CORBA.TRANIENT exception. Review the details of the exception, and take any further action that is necessary.

CREATE_LISTENER_FAILED CREATE_LISTENER_FAILED

Explanation: An exception occurred while a TCP/IP listener was being created.

LOCATE_UNKNOWN_OBJECT • UNSPECIFIED_MARSHAL_25

System action: org.omg.CORBA.INTERNAL

User response: The details of the caught exception are written to the FFDC log. Review the details of the exception, and take any further action that is necessary.

LOCATE_UNKNOWN_OBJECT LOCATE_UNKNOWN_OBJECT

Explanation: The server has no knowledge of the object for which the client has asked in a locate request.

System action: org.omg.CORBA.OBJECT_NOT_EXIST

User response: Ensure that the remote object that is requested resides in the specified server and that the remote reference is up-to-date.

NULL_PI_NAME NULL_PI_NAME

Explanation: One of the following methods has been called:

```
org.omg.PortableInterceptor.ORBInitInfoOperations.  
add_ior_interceptor
```

```
org.omg.PortableInterceptor.ORBInitInfoOperations.  
add_client_request_interceptor
```

```
org.omg.PortableInterceptor.ORBInitInfoOperations  
.add_server_request_interceptor
```

The name() method of the interceptor input parameter returned a null string.

System action: org.omg.CORBA.BAD_PARAM

User response: Change the interceptor implementation so that the name() method returns a non-null string. The name attribute can be an empty string if the interceptor is anonymous, but it cannot be null.

ORB_CONNECT_ERROR_6 ORB_CONNECT_ERROR_6

Explanation: A servant failed to connect to a server-side ORB.

System action: org.omg.CORBA.OBJ_ADAPTER

User response: See the FFDC log for the cause of the problem, then try restarting the application.

POA_DISCARDING POA_DISCARDING

Explanation: The POA Manager at the server is in the discarding state. When a POA manager is in the discarding state, the associated POAs discard all incoming requests (for which processing has not yet begun). For more details, see the section that describes the POAManager Interface in the <http://www.omg.org/cgi-bin/doc?formal/99-10-07>.

System action: org.omg.CORBA.TRANSIENT

User response: Put the POA Manager into the active

state if you want requests to be processed.

RESPONSE_INTERRUPTED RESPONSE_INTERRUPTED

Explanation: The client has enabled the AllowUserInterrupt property and has called for an interrupt on a thread currently waiting for a reply from a remote method call.

System action: org.omg.CORBA.NO_RESPONSE

User response: None.

TRANS_NC_LIST_GOT_EXC TRANS_NC_LIST_GOT_EXC

Explanation: An exception was caught in the NameService while the NamingContext.List() method was executing.

System action: org.omg.CORBA.INTERNAL

User response: The details of the caught exception are written to the FFDC log. Review the details of the original exception, and any further action that is necessary.

UNEXPECTED_CHECKED_EXCEPTION UNEXPECTED_CHECKED_EXCEPTION

Explanation: An unexpected checked exception was caught during the servant_preinvoke method. This method is called before a locally optimized operation call is made to an object of type class. This exception does not occur if the ORB and any Portable Interceptor implementations are correctly installed. It might occur if, for example, a checked exception is added to the Request interceptor operations and these higher level interceptors are called from a back level ORB.

System action: org.omg.CORBA.UNKNOWN

User response: The details of the caught exception are written to the FFDC log. Check whether the class from which it was thrown is at the expected level.

UNSPECIFIED_MARSHAL_25 UNSPECIFIED_MARSHAL_25

Explanation: This error can occur at the server side while the server is reading a request, or at the client side while the client is reading a reply. Possible causes are that the data on the wire is corrupted, or the server and client ORB are not communicating correctly. Communication problems can be caused when one of the ORBs has an incompatibility or bug that prevents it from conforming to specifications.

System action: org.omg.CORBA.MARSHAL

User response: Check whether the IIOP levels and CORBA versions of the client and server are compatible. Try disabling fragmentation (set

com.ibm.CORBA.FragmentationSize to zero) to determine whether it is a fragmentation problem. In this case, analysis of CommTraces (com.ibm.CORBA.CommTrace) might give extra information.

Appendix B. Environment variables

This appendix describes the use of environment variables. Environment variables are overridden by command-line arguments. Where possible, you should use command-line arguments rather than environment variables.

The following information about environment variables is provided:

- “Displaying the current environment”
- “Setting an environment variable”
- “Separating values in a list” on page 414
- “JVM environment settings” on page 414
- “z/OS environment variables” on page 418

Displaying the current environment

This description describes how to show the current environment and how to show an environment variable.

To show the current environment, run:

```
set (Windows)
env (UNIX)
set (z/OS)
WRKENVVAR (i5/OS command prompt)
env (i5/OS qsh or qp2term)
```

To show a particular environment variable, run:

```
echo %ENVNAME% (Windows)
echo $ENVNAME (UNIX, z/OS and I5/OS)
```

Use values exactly as shown in the documentation. The names of environment variables are case-sensitive in UNIX but not in Windows.

Setting an environment variable

This section describes how to set an environment variable and how long a variable remains set.

To set the environment variable **LOGIN_NAME** to *Fred*, run:

```
set LOGIN_NAME=Fred (Windows)
export LOGIN_NAME=Fred (UNIX ksh or bash shells and i5/OS)
```

These variables are set only for the current shell or command-line session.

Separating values in a list

The separator between values is dependant on the platform.

If the value of an environment variable is to be a list:

- On UNIX, i5/OS, and z/OS the separator is typically a colon (:).
- On Windows the separator is typically a semicolon (;).

JVM environment settings

This section describes common environment settings. The categories of settings are general options, deprecated JIT options, Javadump and Heapdump options, and diagnostic options.

General options

The following list summarizes common options. It is not a definitive guide to all the options. Also, the behavior of individual platforms might vary. See individual sections for a more complete description of behavior and availability of these variables.

CLASSPATH=*<directories and archive or compressed files>*

Set this variable to define the search path for application classes and resources. The variable can contain a list of directories for the JVM to find user class files and paths to individual Java archive or compressed files that contain class files; for example, /mycode:/utils.jar (UNIX or i5/OS), D:\mycode;D:\utils.jar (Windows).

Any class path that is set in this way is replaced by the **-cp** or **-classpath** Java argument if used.

IBM_JAVA_COMMAND_LINE

This variable is set by the JVM after it starts. Using this variable, you can find the command-line parameters set when the JVM started.

This setting is not available if the JVM is invoked using JNI.

IBM_JAVA_OPTIONS=*<option>*

Set this variable to store default Java options including **-X**, **-D** or **-verbose:gc** style options; for example, **-Xms256m -Djava.compiler**.

Any options set are overridden by equivalent options that are specified when Java is started.

This variable does not support **-fullversion** or **-version**.

If you specify the name of a trace output file either directly, or indirectly, using a properties file, the output file might be accidentally overwritten if you run utilities such as the trace formatter, dump extractor, or dump viewer. For information about avoiding this problem, see “Controlling the trace” on page 287, Note these restrictions.

JAVA_ASSISTIVE={ **OFF** | **ON** }

Set the **JAVA_ASSISTIVE** environment variable to **OFF** to prevent the JVM from loading Java Accessibility support.

JAVA_FONTS=*<list of directories>*

Set this environment variable to specify the font directory. Setting this variable is equivalent to setting the properties **java.awt.fonts** and **sun.java2d.fontpath**.

JAVA_HIGH_ZIPFDS=<value>

The X Window System cannot use file descriptors above 255. Because the JVM holds file descriptors for open jar files, X can run out of file descriptors. As a workaround, set the **JAVA_HIGH_ZIPFDS** environment variable to tell the JVM to use higher file descriptors for jar files. Set it to a value in the range 0 - 512. The JVM then opens the first jar files using file descriptors up to 1024. For example, if your program is likely to load 300 jar files:

```
export JAVA_HIGH_ZIPFDS=300
```

The first 300 jar files are then loaded using the file descriptors 724 - 1023. Any jar files opened after that are opened in the typical range.

This variable is for Linux only.

JAVA_MMAP_MAXSIZE=<size>

Set this environment variable to specify a maximum size in MB. Compressed or jar files smaller than this size are opened with memory mapping. Files larger than this size are opened with normal I/O.

The default size is 0. This default disables memory mapping.

JAVA_PLUGIN_AGENT=<version>

Set this variable to specify the version of Mozilla.

This variable is for Linux and z/OS only.

JAVA_PLUGIN_REDIRECT=<value>

Set this variable to a non-null value to redirect JVM output, while serving as a plug-in, to files. The standard output is redirected to the file `plugin.out`. The error output is redirected to the file `plugin.err`.

This variable is for Linux and z/OS only.

JAVA_ZIP_DEBUG=<value>

Set this variable to any value to display memory map information as it is created.

LANG=<locale>

Set this variable to specify a locale to use by default.

This variable is for AIX, Linux, and z/OS only.

LD_LIBRARY_PATH=<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 Linux 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.

SYS_LIBRARY_PATH=<path>

Set this variable to define the library path.

This variable is for Linux and z/OS only.

Deprecated JIT options

The following list describes deprecated JIT options:

IBM_MIXED_MODE_THRESHOLD

Use **-Xjit:count=<value>** instead of this variable.

JAVA_COMPILER

Use **-Djava.compiler=<value>** instead of this variable.

Javadump and Heapdump options

The following list describes the Javadump and Heapdump options. The recommended way of controlling the production of diagnostic data is the **-Xdump** command-line option, described in Chapter 21, “Using dump agents,” on page 223.

DISABLE_JAVADUMP={ TRUE | FALSE }

This variable disables Javadump creation when set to TRUE.

Use the command-line option **-Xdisablejavadump** instead. Avoid using this environment variable because it makes it more difficult to diagnose failures.

On z/OS, use **JAVA_DUMP_OPTS** in preference.

IBM_HEAPDUMP or IBM_HEAP_DUMP={ TRUE | FALSE }

These variables control the generation of a Heapdump.

When the variables are set to 0 or FALSE, Heapdump is not available. When the variables are set to anything else, Heapdump is enabled for crashes or user signals. When the variables are not set, Heapdump is not enabled for crashes or user signals.

IBM_HEAPDUMP_OUTOFMEMORY={ TRUE | FALSE }

This variable controls the generation of a Heapdump when an out-of-memory exception is thrown.

When the variable is set to TRUE or 1 a Heapdump is generated each time an out-of-memory exception is thrown, even if it is handled. When the variable is set to FALSE or 0, a Heapdump is not generated for an out-of-memory exception. When the variable is not set, a Heapdump is generated when an out-of-memory exception is not caught and handled by the application.

IBM_HEAPDUMPPDIR=<directory>

This variable specifies an alternative location for Heapdump files.

On z/OS, **_CEE_DMPTARG** is used instead.

IBM_JAVACOREDIR=<directory>

This variable specifies an alternative location for Javadump files; for example, on Linux **IBM_JAVACOREDIR=/dumps**

On z/OS, **_CEE_DMPTARG** is used instead.

IBM_JAVADUMP_OUTOFMEMORY={ TRUE | FALSE }

This variable controls the generation of a Javadump when an out-of-memory exception is thrown.

When the variable is set to TRUE or 1, a Javadump is generated each time an out-of-memory exception is thrown, even if it is handled. When the variable is set to FALSE or 0, a Javadump is not generated for an out-of-memory exception. When the variable is not set, a Javadump is generated when an out-of-memory exception is not caught and handled by the application.

IBM_NOSIGHANDLER={ TRUE }

This variable disables the signal handler when set to any value. If no value is supplied, the variable has no effect and the signal handler continues to work.

The variable is equivalent to the command-line option **-Xrs:all**

JAVA_DUMP_OPTS=<value>

This variable controls how diagnostic data are dumped.

For a fuller description of **JAVA_DUMP_OPTS** and variations for different platforms, see “Dump agent environment variables” on page 239.

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. This variable defaults to C:\Temp on Windows.

Diagnostics options

The following list describes the diagnostics options:

IBM_COREDIR=<directory>

Set this variable to specify an alternative location for system dumps and snap trace.

On z/OS, **_CEE_DMPTARG** is used instead for snap trace, and transaction dumps are written to TSO according to **JAVA_DUMP_TDUMP_PATTERN**.

On Linux, the dump is written to the OS specified directory, before being moved to the specified location.

IBM_JVM_DEBUG_PROG=<debugger>

Set this variable to start the JVM under the specified debugger.

This variable is for Linux only.

IBM_MALLOCTRACE=TRUE

Setting this variable to a non-null value lets you trace memory allocation in the JVM. You can use this variable with the **-Dcom.ibm.dbgmalloc=true** system property to trace native allocations from the Java classes.

This variable is equivalent to the command-line option **-memorycheck**.

IBM_USE_FLOATING_STACKS=TRUE

Set this variable to override the automatic disabling of floating stacks. See the *Linux SDK and Runtime User Guide*. If this variable is not set, the launcher might set **LD_ASSUME_KERNEL=2.2.5**.

This variable is for Linux only

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/InternalServerError`.

A Signal 11 is followed by a JVMXE message and then the JVM terminates.

JAVA_PLUGIN_TRACE=TRUE

When this variable is set to TRUE or 1, a Java plug-in trace is produced for the session when an application runs. Traces are produced from both the Java and Native layer.

By default, this variable is set to FALSE, so that a Java plug-in trace is not produced.

z/OS environment variables

This section describes the environment variables of the z/OS JVM.

IBM_JAVA_ABEND_ON_FAILURE=Y

Tells the Java launcher to mark the Task Control Block (TCB) with an abend code if the JVM fails to load or is terminated by an uncaught exception. By default, the Java launcher will not mark the TCB.

JAVA_DUMP_OPTS

See Chapter 23, “Using Heapdump,” on page 257 for details.

JAVA_DUMP_TDUMP_PATTERN=string

Result: The specified string is passed to IEATDUMP to use as the data/set name for the Transaction Dump. The default string is:

```
%uid.JVM.TDUMP.%job.D%y%m%d.T%H%M%S
```

where %uid is found from the following C code fragment:

```
pwd = getpwuid(getuid());  
pwd->pw_name;
```

JAVA_LOCAL_TIME

The z/OS JVM does not look at the offset part of the TZ environment variable and will therefore incorrectly show the local time. Where local time is not GMT, you can set the environment variable

JAVA_LOCAL_TIME to display the correct local time as defined by TZ.

JAVA_THREAD_MODEL

JAVA_THREAD_MODEL can be defined as one of:

NATIVE

JVM uses the standard, POSIX-compliant thread model that is provided by the JVM. All threads are created as **_MEDIUM_WEIGHT** threads.

HEAVY

JVM uses the standard thread package, but all threads are created as **_HEAVY_WEIGHT** threads.

MEDIUM

Same as **NATIVE**.

NULL

Default case: Same as **NATIVE/MEDIUM**.

Appendix C. Messages

This appendix lists error messages in numeric sequence.

These messages, error codes, and exit codes are generated by the JVM.

If the JVM fills all available memory, it might not be able to produce a message and a description for the error that caused the problem. Under such a condition only the message might be produced.

From Java 5 SR10 selected messages are routed to the system log and also written to stderr or stdout. This message logging feature is enabled by default. To disable message logging use the **-Xlog:none** option. The specific messages that are logged are JVMDUMP032I, which is issued when dumps are produced by the JVM, and JVMDMP025I, which is a z/OS dump failure message.

Logged messages can be found in the different locations, according to the platform.

- On AIX, messages are logged by the syslog daemon. By default, the syslog daemon does not run, therefore you must start it manually. You can redirect messages from the syslog daemon to the AIX error log facility by performing the following configuration steps:
 1. Set up a redirect in the file `syslog.conf` so that syslog messages are sent to the error log, by adding the following line:
`user.debug errlog`
 2. If syslogd is already running, reload the updated configuration by running the following command:
`refresh -s syslogd`
 3. The updated configuration is used each time syslogd starts.
 4. Use the AIX `errpt` command or the System Management Interface Tool (SMIT) to read the messages sent to the error log.

If you do not enable syslog to errlog redirection, logged messages go into the default syslog file. If syslogd is not running, logged messages are lost.

For more information about AIX logging, see: General Programming Concepts: Writing and Debugging Programs

- On Linux, messages are logged by the syslog daemon. To find where messages are logged, check the syslog configuration file.
- On Windows, messages are logged in the application events section of the event viewer.
- On z/OS, messages are sent to the operator console. To see the messages, go from the **ispf** panel to the **sdsf** panel, then open the **log** panel.

The message appendix is not complete. It will be enlarged in future editions.

The messages are listed in:

- “DUMP messages” on page 420
- “J9VM messages” on page 424
- “SHRC messages” on page 427

DUMP messages

Dump agent messages.

JVMDUMP000E Dump option unrecognized: -Xdump:%s

Explanation: An option used with the -Xdump parameter is unknown.

System action: The JVM terminates.

User response: Use -Xdump:help to find the correct syntax for the -Xdump options.

Related information

“Using the -Xdump option” on page 223

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

JVMDUMP001E Dump event unrecognized: ...%s

Explanation: The event name provided with the -Xdump parameter is unknown.

System action: The JVM terminates.

User response: Use -Xdump:events to find the supported set of event names.

Related information

“Dump events” on page 232

Dump agents are triggered by events occurring during JVM operation.

JVMDUMP002W Token unrecognized: %%1\$c

Explanation: An unidentified token was found in the dump label.

System action: The JVM terminates.

User response: Use -Xdump:tokens to find the supported set of tokens.

JVMDUMP003E Dump agent unrecognized: %s

Explanation: An unsupported dump type has been requested.

System action: The JVM terminates.

User response: Use -Xdump:help to find the supported set of dump types.

Related information

“Dump agents” on page 227

A dump agent performs diagnostic tasks when triggered. Most dump agents save information on the state of the JVM for later analysis. The “tool” agent can be used to trigger interactive diagnostics.

JVMDUMP004E Missing file name

Explanation: The dump file name could not be found.

System action: The dump cannot be written.

User response: Provide a valid filename for the dump.

Related information

“file option” on page 234

The file option is used by dump agents that write to a file.

JVMDUMP005E Missing external tool

Explanation: The executable file for the tool could not be found.

System action: The tool cannot run.

User response: Provide a valid path and filename for the executable file.

Related information

“exec option” on page 233

The exec option is used by the tool dump agent to specify an external application to start.

JVMDUMP006I Processing dump event \"%1\$s\", detail \"%3\$.*2\$s\" - please wait.

Explanation: A dump event has occurred and is being handled.

System action: The JVM attempts to process the event.

User response: No response is required.

Related information

Chapter 21, “Using dump agents,” on page 223

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.

JVMDUMP007I JVM Requesting %1\$s dump using '%2\$s'

Explanation: The JVM is about to write a dump because either an event (such as an exception being thrown) was triggered or the user requested the dump through JVMTI, JVMRI, -Xtrace:trigger or the com.ibm.jvm.Dump Java API.

System action: The JVM will attempt to write the dump. A second message will be printed when the dump has been written.

User response: Once the dump is written the user should review the dump and take appropriate action.

Related information

Chapter 21, “Using dump agents,” on page 223
 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.

JVMDUMP008I Using '%s'

Explanation: THIS MESSAGE IS RESERVED FOR FUTURE USE

System action: THIS MESSAGE IS RESERVED FOR FUTURE USE

User response: THIS MESSAGE IS RESERVED FOR FUTURE USE

JVMDUMP009E %s dump not available

Explanation: The specified dump type is not supported on this platform.

System action: The JVM does not write the dump.

User response: Use **-Xdump:what** to list the supported dump types.

JVMDUMP010I %1\$s dump written to %2\$s

Explanation: The dump was written to the specified location.

System action: The JVM continues.

User response: To understand how to interpret a Java dump file, refer to diagnostic information.

JVMDUMP011I %1\$s dump created process %2\$d

Explanation: A tool dump process has been created.

System action: The JVM runs the executable process specified for the tool dump. Any parameters are passed to the process.

User response: Refer to the documentation for the tool creating the dump process.

Related information

“Tool option” on page 230

The **tool** option allows external processes to be started when an event occurs.

JVMDUMP012E Error in %1\$s dump:%2\$s

Explanation: The JVM detected an error while attempting to produce a dump.

System action: The JVM continues, but the dump might not be usable.

User response: Refer to diagnostic information or contact your local IBM support representative.

JVMDUMP013I Processed dump event \"%1\$s\", detail \"%3\$.*2\$s\".

Explanation: A dump event occurred and has been handled.

System action: The JVM continues.

User response: Refer to other messages issued by the JVM for the location of the dump file, or for other actions required.

JVMDUMP014E VM Action unrecognized: ...%s

Explanation: A specified dump request action was not understood by the JVM.

System action: The JVM produces help information and terminates.

User response: Use **-Xdump:request** to check that your request is valid.

Related information

“Using the -Xdump option” on page 223

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

JVMDUMP015I Aborting: Cannot open or read (%s)

Explanation: The JVM cannot open a dump file in read mode.

System action: INTERNAL USE ONLY

User response: INTERNAL USE ONLY

JVMDUMP016I Aborting: Cannot create file (%s)

Explanation: An attempt by the JVM to open a new file has failed.

System action: INTERNAL USE ONLY

User response: INTERNAL USE ONLY

JVMDUMP017I Aborting: Cannot compress file (%s)

Explanation: The JVM cannot compress a file.

System action: INTERNAL USE ONLY

User response: INTERNAL USE ONLY

JVMDUMP018W Requested event is not available: run with -Xdump:dynamic flag

Explanation: A dump has been requested on an event type that is not supported.

System action: INTERNAL USE ONLY

User response: INTERNAL USE ONLY

JVMDUMP019I JVM requesting %s dump

Explanation: A dump file of the type specified has been requested.

System action: The JVM attempts to produce a dump of the specified type.

User response: Wait for a message indicating that the dump is complete.

JVMDUMP020I %s dump has been written

Explanation: A dump file of the type specified has been written.

System action: The JVM continues.

User response: To understand how to interpret the dump file, refer to Java diagnostic information.

JVMDUMP021W The requested heapdump has not been produced because exclusive access was not requested or could not be obtained.

Explanation: THIS MESSAGE IS RESERVED FOR FUTURE USE

System action: THIS MESSAGE IS RESERVED FOR FUTURE USE

User response: THIS MESSAGE IS RESERVED FOR FUTURE USE

JVMDUMP022W The requested heap compaction has not been performed because exclusive access was not requested or could not be obtained.

Explanation: The garbage collector could not run because the gc thread did not have exclusive access to the heap.

System action: The dump file is not produced.

User response: Modify the **-Xdump** option to request exclusive access. See the Java diagnostic information for valid dump request types.

JVMDUMP023W The requested heap preparation has not been performed because exclusive access was not requested or could not be obtained.

Explanation: The dump thread must lock the heap to prevent changes while the dump is taken. Either no attempt was made to obtain the lock or the lock could not be obtained.

System action: The JVM does not produce the specified dump.

User response: Modify the **-Xdump** option to request

exclusive access. See the Java diagnostic information for valid dump request types.

JVMDUMP024W Multiple heapdumps were requested but %%id is missing from file label:. Dumps will overwrite

Explanation: The JVM replaces the %%id insertion point with a unique number. Because %%id is missing the replacement cannot take place, causing a potential file name clash and file overwrite.

System action: The JVM informs the user and produces dumps as requested.

User response: If you do not want files overwritten, specify a file label that includes the %%id.

JVMDMP025I IEATDUMP failure for DSN='%s' RC=0x%08X RSN=0x%08X

Explanation: An IEATDUMP was requested but could not be produced.

System action: The JVM will output the message on the operator console.

User response: Check the response code provided in the error using the Diagnostics Guide or the z/OS V1R7.0 MVS Authorized Assembler Services Reference, 36.1.10 Return and Reason Codes.

Related information

“Setting up dumps” on page 150

The JVM generates a Javdump and System Transaction Dump (SYSTDUMP) when particular events occur.

JVMDUMP026 IEATDUMP Name exceeding maximum allowed length. Default name used.

Explanation: The file label exceeded the maximum length for file names on z/OS.

System action: The dump file is not produced.

User response: Refer to Java diagnostic information for information about producing dumps on z/OS.

JVMDUMP027W The requested heapdump has not been produced because another component is holding the VM exclusive lock.

Explanation: The exclusive VM lock must be held to produce a usable heapdump. Although the VM exclusive lock was requested by the user, the VM could not immediately take the lock and has given up rather than risk a deadlock.

System action: The VM will not produce a heapdump.

User response: The component that is holding the VM exclusive lock will probably release it in a short period of time. Try taking the heapdump again after a minute.

JVMDUMP028W The VM exclusive lock could not be acquired before taking the system dump.

Explanation: The user requested that the exclusive VM lock be taken before taking a system dump. However, when the dump was triggered another component was holding the lock and, rather than risk a deadlock, the VM is continuing without the lock.

System action: The VM will write a system dump without taking the VM exclusive lock. This may mean the dump shows an inconsistent view of the VM data structures and heap.

User response: The component that is holding the VM exclusive lock will probably release it in a short period of time. Try taking the system dump again after a minute.

JVMDUMP029W The request for prepwalk or compact before taking a system dump will be ignored because the VM exclusive lock was not requested.

Explanation: The user requested the prepwalk or compact options before taking a system dump, but did not request the VM exclusive lock. These actions require the exclusive VM lock.

System action: The system dump is taken without running prepwalk or compact.

User response: Modify the `-Xdump:system` parameter to include the exclusive request option. For example:
`-Xdump:system:events=user,request=exclusive+compact+prepwalk`

JVMDUMP030W Cannot write dump to file %s: %s

Explanation: The JVM was unable to write a dump to the specified file. There might be multiple causes, including insufficient file system permissions or specifying a file that exists.

System action: The JVM uses a default file name. The name and location of the file are indicated by the messages produced when the dump is written.

User response: Correct the problem with the specified file path or change the target dump file with the `file=` option.

JVMDUMP031W The requested heapdump has not been produced because the VM exclusive lock was not requested. Add request=exclusive+prepwalk+compact to your -Xdump:heap: command line option.

Explanation: A heapdump dump agent was configured using the `-Xdump` option or JVMTI without requesting that the exclusive VM lock.

System action: The JVM does not take the heapdump

because the dump might be corrupted without the VM exclusive lock in place.

User response: Change the `-Xdump:heap:` option to include the `request=exclusive` option. For example:
`-Xdump:heap:events=user,request=exclusive`

JVMDUMP032I JVM requested %1\$s dump using '%2\$s' in response to an event

Explanation: The JVM writes a dump because an event, such as an exception, was triggered.

System action: The JVM attempts to write the dump. A second message is produced when the dump is complete.

User response: Review the dump and take appropriate action.

JVMDUMP033I JVM requested %1\$s dump in response to an event

Explanation: The JVM writes a dump because an event, such as an exception, has been triggered.

System action: The JVM attempts to write the dump. A second message is produced when the dump is complete.

User response: Review the dump and take appropriate action.

JVMDUMP034I User requested %1\$s dump using '%2\$s' through %3\$s

Explanation: The JVM writes a dump in response to a request through an API, such as JVMTI, or through the `-Xtrace:trigger` option.

System action: The JVM attempts to write the dump. A second message is produced when the dump is complete.

User response: Review the dump and take appropriate action.

JVMDUMP035I User requested %1\$s dump through %2\$s

Explanation: The JVM writes a dump in response to a user request through an API, such as JVMTI, or through the `-Xtrace:trigger` option.

System action: The JVM attempts to write the dump. A second message is produced when the dump is complete.

User response: Review the dump and take appropriate action.

JVMDUMP036I Invalid or missing -Xdump filter

Explanation: A valid -Xdump filter must be supplied.

System action: The dump agent is not loaded.

User response: Modify the dump option to include a valid filter.

JVMDUMP037E Error in %1\$s dump: %2\$s failed, error code: %3\$d

Explanation: An error has occurred in a JVM dump agent. An operating system call used by the agent has failed.

System action: The JVM continues.

User response: Check preceding JVM messages. The JVMDUMP007I message includes the command string supplied for tool dumps. Check that the command string is correct.

JVMDUMP038 Snap dump is not written because tracing to file: %1\$s

Explanation: Snap dumps are not produced when a trace output file is in use.

System action: The JVM continues.

User response: Use the trace file specified in the message.

J9VM messages

JVMJ9VM000 Malformed value for IBM_JAVA_OPTIONS

Explanation: JVM Initialisation is using the environment variable IBM_JAVA_OPTIONS and has found an error during parsing. Errors such as unmatched quotes can give rise to this.

System action: The JVM terminates.

User response: Check the syntax of the environment variable IBM_JAVA_OPTIONS

User response: Correct the -Djava.compiler specification and retry.

JVMJ9VM004E Cannot load library required by: %s

Explanation: JVM initialization uses system services to load numerous libraries (some of which are user specified), often these libraries have dependencies on other libraries. If, for various reasons, a library cannot be loaded then this message is produced.

System action: The JVM terminates.

User response: Check your system to ensure that the indicated libraries are available and accessible. If the failing application has been used successfully then a recent environment change to your system is a likely cause. If the failure persists then contact your IBM Service representative.

JVMJ9VM001 Malformed value for -Xservice

Explanation: JVM Initialisation is attempting to use the specified -Xservice option and found a parsing error. Errors such as unmatched quotes can give rise to this.

System action: The JVM terminates.

User response: Check the syntax of the -Xservice option

JVMJ9VM005E Invalid value for environment variable: %s

Explanation: The identified environment variable has been given an invalid value.

System action: The JVM terminates.

User response: Correct the specified environment variable and retry.

JVMJ9VM002 Options file not found

Explanation: The options file specified using -Xoptionsfile couldn't be found.

System action: The JVM terminates.

User response: Correct the -Xoptionsfile option on the commandline and retry.

JVMJ9VM006E Invalid command-line option: %s

Explanation: The identified command line option is invalid.

System action: The JVM terminates.

User response: Correct or remove the command line option and retry.

JVMJ9VM003 JIT compiler "%s" not found. Will use interpreter.

Explanation: The value specified using the -Djava.compiler option is not valid.

System action: The JVM continues but without a compiler (note this is generally slower than with a compiler).

JVMJ9VM007E Command-line option unrecognised: %s

Explanation: The identified command line option is not recognised as valid.

System action: The JVM terminates.

User response: Correct or remove the command line option and retry.

JVMJ9VM008 J9VMDllMain not found

Explanation: J9VMDllMain is the main module entry point for system libraries (dlls). If not found then the module is unusable.

System action: The JVM Terminates.

User response: Contact your IBM Service representative.

JVMJ9VM009 J9VMDllMain failed

Explanation: J9VMDllMain is the main module entry point for system libraries (dlls). There has been a failure in its use.

System action: The JVM Terminates.

User response: Contact your IBM Service representative.

JVMJ9VM010W Failed to initialize %s

Explanation: The identified library couldn't be initialized. This message is generally associated with JVMPI functionality.

System action: The JVM continues, however the expected functionality may be affected.

User response: Contact your IBM Service representative.

JVMJ9VM011W Unable to load %s: %s

Explanation: The JVM attempted to load the library named in the first parameter, but failed. The second parameter gives further information on the reason for the failure.

System action: The JVM continues, however if the library contained JVM core functionality then the JVM may terminate later (after issuing further messages).

User response: Check your system to ensure that the indicated libraries are available and accessible. If the problem persists then contact your IBM Service representative.

JVMJ9VM012W Unable to unload %s: %s

Explanation: The JVM attempted to unload the library named in the first parameter, but failed. The second parameter gives further information on the reason for the failure.

System action: The JVM continues.

User response: If the problem persists then contact your IBM Service representative.

JVMJ9VM013W Initialization error in function

%s(%d): %s

Explanation: This will generally be an internal error within the JVM.

System action: The JVM continues, but if the error is in a critical area then the JVM will probably terminate after issuing further messages.

User response: If the problem persists then contact your IBM Service representative.

JVMJ9VM014W Shutdown error in function %s(%d):

%s

Explanation: During shutdown processing an internal error was detected (identified further in the message).

System action: The JVM continues.

User response: If the problem persists then contact your IBM Service representative.

JVMJ9VM015W Initialization error for library

%s(%d): %s

Explanation: During JVM Initialization various libraries (aka dlls) are loaded and initialized. If something goes wrong during this initialization this message is produced. Usually this reflects errors in JVM invocation such as invalid option usage which will normally have given rise to other messages.

System action: The JVM terminates.

User response: This message is often seen as a follow-on to other messages indicating the problem that caused initialization of this library to fail. Correct the problem(s) indicated by previous messages and retry.

JVMJ9VM016W Shutdown error for library %s(%d):

%s

Explanation: During shutdown processing an internal error was detected (identified further in the message).

System action: The JVM continues.

User response: If the problem persists then contact your IBM Service representative.

JVMJ9VM017 Could not allocate memory for command line option array

Explanation: During JVM initialization the command line options are stored in memory. The JVM couldn't obtain sufficient memory from the system to complete the process.

System action: The JVM terminates.

User response: This is unlikely to be a problem caused by the JVM invocation. Check your machine for other hardware/software faults and retry (after restart of the machine). If the problem persists then contact your IBM Service representative.

JVMJ9VM018 Could not allocate memory for DLL load table pool

Explanation: This error is issued when memory could not be allocated to expand an internal table. It is likely that this error is external to the JVM and may be a Operating System or machine problem.

System action: The JVM continues, however it is exceedingly likely that the JVM will fail soon.

User response: Check your machine for other problems, you may need to reboot and retry. If the problem persists then contact your IBM Service representative.

JVMJ9VM032 Fatal error: unable to load %s: %s

Explanation: A required library couldn't be loaded. The first parameter gives the name of the library and the second more details on the reasons why it could not be loaded.

System action: The JVM terminates.

User response: Check that the library exists in the requisite place and has the correct access levels. If the problem persists then contact your IBM Service representative.

JVMJ9VM033 Fatal error: failed to initialize %s

Explanation: A required library couldn't be initialized. The parameter gives the name of the library.

System action: The JVM terminates.

User response: If the problem persists then contact your IBM Service representative.

JVMJ9VM035 Unable to allocate OutOfMemoryError

Explanation: The JVM tried to issue an OutOfMemoryError but failed. This is often indicative of a failure in the user application design/useage.

System action: The JVM Terminates.

User response: Check the memory usage of your application and retry (possibly using the -Xmx option on startup). If the problem persists then contact your IBM Service representative.

JVMJ9VM038E -Xthr: unrecognized option --' %s'

Explanation: The JVM has been invoked with an unrecognized -Xthr option.

System action: The JVM Terminates.

User response: Correct or remove the invalid -Xthr option and retry.

JVMJ9VM039 -Xscmx is ignored if -Xshareclasses is not specified

Explanation: The -Xscmx'n' option is not meaningful unless shared class support is active.

System action: The JVM continues.

User response: Remove the -Xscmx option or activate shared classes by using -Xshareclasses.

SHRC messages

JVMSHRC004E Cannot destroy cache "%1\$s"

Explanation: It has not been possible to destroy the named shared classes cache.

System action: Processing continues.

User response: Other messages may have been issued indicating the reason why the cache has not been destroyed. Investigate these messages.

JVMSHRC005 No shared class caches available

Explanation: There are no shared class caches present on the system which can be processed by the command requested

System action: Processing continues.

User response: None required.

JVMSHRC006I Number of caches expired within last %1\$d minutes is %2\$d

Explanation: This is an information message issued by the system.

System action: Processing continues.

User response: None required.

JVMSHRC007I Failed to remove shared class cache "%1\$s"

Explanation: It has not been possible to remove the indicated shared class cache.

System action: Processing continues.

User response: Other messages may have been issued indicating the reason why the cache has not been destroyed. Investigate these messages.

JVMSHRC008I Shared Classes Cache created: %1\$s size: %2\$d bytes

Explanation: This is an information message notifying you that a shared classes cache of the given name and size has been created.

System action: The JVM continues.

User response: None required. This is an information message issued when verbose shared classes messages have been requested.

JVMSHRC009I Shared Classes Cache opened: %1\$s size: %2\$d bytes

Explanation: This is an information message notifying you that an existing shared classes cache of the given name and size has been opened.

System action: The JVM continues.

User response: None required. This is an information message issued when verbose shared classes messages have been requested.

JVMSHRC010I Shared Cache "%1\$s" is destroyed

Explanation: This is an information message notifying you that the named shared classes cache has been destroyed as requested.

System action: A JVM will not be created and a failure message will be issued, however, this is a good normal response when you request a shared classes cache to be destroyed.

User response: None required. This is an information message issued when you request a shared classes cache to be destroyed.

JVMSHRC012I Cannot remove shared cache "%1\$s" as there are JVMs still attached to the cache

Explanation: You have requested that the system destroy a shared classes cache, but a process or processes are still attached to it.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Wait until any other processes using the shared classes cache have terminated and then destroy it.

JVMSHRC013E Shared cache "%1\$s" memory remove failed

Explanation: You have requested that the system destroy a shared classes cache, but it has not been possible to remove the shared memory associated with the cache.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative.

JVMSHRC014E Shared cache "%1\$s" semaphore remove failed

Explanation: You have requested that the system destroy a shared classes cache, but it has not been possible to remove the shared semaphore associated with the cache.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative.

JVMSHRC015 Shared Class Cache Error: Invalid flag

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative.

JVMSHRC017E Error code: %d

Explanation: This message shows the error code relating to a error that will have been the subject of a previous message.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative, unless previous messages indicate a different response.

JVMSHRC018 cannot allocate memory

Explanation: The system is unable to obtain sufficient memory.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low of memory resource, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC019 request length is too small

Explanation: The size requested for the shared classes cache is too small.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Increase the requested size for the shared classes cache using the -Xscmx parameter or allow it to take the default value by not specifying -Xscmx.

JVMSHRC020 An error has occurred while opening semaphore

Explanation: An error has occurred in shared class processing. Further messages may follow providing more detail.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative, unless subsequent messages indicate otherwise.

JVMSHRC021 An unknown error code has been returned

Explanation: An error has occurred in shared class processing. This message should be followed by details of the numeric error code returned.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative.

JVMSHRC022 Error creating shared memory region

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low on resources, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC023E Cache does not exist

Explanation: An attempt has been made to open a shared classes cache which does not exist.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative.

JVMSHRC024 shared memory detach error

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative.

JVMSHRC025 error attaching shared memory

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low on resources, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC026E Cannot create cache of requested size: Please check your SHMMAX and SHMMIN settings

Explanation: The system has not been able to create a shared classes cache of the size required via the -Xscmx parameter (16MB if -Xscmx is not specified).

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Please refer to the User Guide for a discussion of shared memory size limits for your operating system and restart the JVM with an acceptable shared cache size.

JVMSHRC027E Shared cache name is too long

Explanation: The name specified for the shared classes cache is too long for the operating system.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Specify a shorter name for the shared classes cache and restart the JVM.

JVMSHRC028E Permission Denied

Explanation: The system does not have permission to access a system resource. A previous message should be issued indicating the resource that cannot be accessed. For example, a previous message may indicate that there was an error opening shared memory. This message would indicate that the error was that you do not have permission to access the shared memory.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Shared class caches are created so that only the user who created the cache has access to it, unless the -Xshareclasses:groupAccess is specified when other members of the creator's group may also access it. If you do not come into one of these categories, you will not have access to the cache. For more information on permissions and shared classes, see "Chapter 4. Understanding Shared Classes".

JVMSHRC029E Not enough memory left on the system

Explanation: There is not enough memory available to create the shared cache memory or semaphore. A previous message will have indicated which could not be created.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low of memory resource, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC030E The Shared Class Cache you are attaching has invalid header.

Explanation: The shared classes cache you are trying to use is invalid.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative.

JVMSHRC031E The Shared Class Cache you are attaching has incompatible JVM version.

Explanation: The shared classes cache you are trying to use is incompatible with this JVM.

System action: The JVM terminates.

User response: If the requested shared classes cache is no longer required, destroy it and rerun. If it is still required, for example, by another process running a different level of the JVM, create a new cache by specifying a new name to the JVM using the `-Xshareclasses:name`

JVMSHRC032E The Shared Class Cache you are attaching has wrong modification level.

Explanation: The shared classes cache you are trying to use is incompatible with this JVM.

System action: The JVM terminates.

User response: If the requested shared classes cache is no longer required, destroy it and rerun. If it is still required, for example, by another process running a different level of the JVM, create a new cache by specifying a new name to the JVM using the `-Xshareclasses:name`

JVMSHRC057 Wrong parameters for expire option

Explanation: The value specified for the expire parameter of `-Xshareclasses` is invalid.

System action: The JVM terminates, unless you have specified the nonfatal option with `"-Xshareclasses:nonfatal"`, in which case the JVM continues without using Shared Classes.

User response: Rerun the command with a valid expire value. This must be a positive integer.

JVMSHRC058 ERROR: Cannot allocate memory for ClasspathItem in shrinit::hookStoreSharedClass

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with `"-Xshareclasses:nonfatal"`, in which case the JVM continues without using Shared Classes.

User response: The system may be low of memory resource, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC059 ERROR: Cannot allocate memory for ClasspathItem in shrinit::hookFindSharedClass

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with `"-Xshareclasses:nonfatal"`, in which case the JVM continues without using Shared Classes.

User response: The system may be low of memory resource, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC060 ERROR: Cannot allocate memory for string buffer in shrinit::hookFindSharedClass

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with `"-Xshareclasses:nonfatal"`, in which case the JVM continues without using Shared Classes.

User response: The system may be low of memory resource, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC061 Cache name should not be longer than 64 chars. Cache not created.

Explanation: The name of the shared classes cache specified to the JVM exceeds the maximum length.

System action: The JVM terminates, unless you have specified the nonfatal option with `"-Xshareclasses:nonfatal"`, in which case the JVM continues without using Shared Classes.

User response: Change the requested shared classes cache name so that it is shorter than the maximum allowed length.

JVMSHRC062 ERROR: Error copying username into default cache name

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with `"-Xshareclasses:nonfatal"`, in which case the JVM continues without using Shared Classes.

User response: This may be due to problems with the operating system, please retry. If the situation persists, contact your IBM service representative.

JVMSHRC063 ERROR: Cannot allocate memory for sharedClassConfig in shrinit

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with `"-Xshareclasses:nonfatal"`, in which case the JVM continues without using Shared Classes.

User response: The system may be low of memory

resource, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

**JVMSHRC064 ERROR: Failed to create
configMonitor in shrinit**

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: This may be due to problems with the operating system, please retry. If the situation persists, contact your IBM service representative.

**JVMSHRC065 ERROR: Cannot allocate pool in
shrinit**

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low of memory resource, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

**JVMSHRC066 INFO: Locking of local hashtables
disabled**

Explanation: This message confirms that locking of local hashtables for the shared classes cache has been disabled as requested. It is only issued when verbose messages are requested.

System action:

User response:

JVMSHRC067 INFO: Timestamp checking disabled

Explanation: This message confirms that shared classes timestamp checking has been disabled as requested. It is only issued when verbose messages are requested.

System action: The JVM continues.

User response: None required.

**JVMSHRC068 INFO: Local cacheing of classpaths
disabled**

Explanation: This message indicates that, when requested, cacheing of classpaths in the shared classes cache has been disabled. This message is only issued

when shared classes verbose messages are requested.

System action: The JVM continues.

User response: None required.

**JVMSHRC069 INFO: Concurrent store contention
reduction disabled**

Explanation: This message confirms that shared classes concurrent store contention reduction has been disabled as requested. It is only issued when verbose messages are requested.

System action: The JVM continues.

User response: None required.

JVMSHRC070 INFO: Incremental updates disabled

Explanation: This message confirms that shared classes incremental updates have been disabled as requested. It is only issued when verbose messages are requested.

System action: The JVM continues.

User response: None required.

**JVMSHRC071 ERROR: Command-line option "%s"
requires sub-option**

Explanation: The specified command-line option requires further information.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Specify the additional information required for the command-line option and rerun.

**JVMSHRC072 ERROR: Command-line option "%s"
unrecognised**

Explanation: The specified command-line option is not recognised.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Correct or remove the invalid command-line option and rerun.

**JVMSHRC077 ERROR: Failed to create
linkedListImpl pool in
SH_ClasspathManagerImpl2**

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with

"-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low on resources, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

**JVMSHRC078 ERROR: Failed to create
linkedListHdr pool in
SH_ClasspathManagerImpl2**

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low on resources, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

**JVMSHRC079 ERROR: Cannot create hashtable in
SH_ClasspathManagerImpl2**

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative.

**JVMSHRC080 ERROR: Cannot allocate memory for
hashtable entry**

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low of memory resource, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

**JVMSHRC081 ERROR: Cannot create cpeTableMutex
in SH_ClasspathManagerImpl2**

Explanation: An error has occurred while initialising shared classes.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM

continues without using Shared Classes.

User response: The system may be low on resources, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

**JVMSHRC082 ERROR: Cannot create
identifiedMutex in
SH_ClasspathManagerImpl2**

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low on resources, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

**JVMSHRC083 ERROR: Cannot allocate memory for
identifiedClasspaths array in
SH_ClasspathManagerImpl2**

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low of memory resource, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

**JVMSHRC084 ERROR: Cannot allocate memory for
linked list item**

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low of memory resource, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

**JVMSHRC085 ERROR: Cannot allocate memory for
linked list item header**

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have

specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low of memory resource, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC086 ERROR: Cannot enter ClasspathManager hashtable mutex

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low on resources, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC087 ERROR: MarkStale failed during ClasspathManager::update()

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative.

JVMSHRC088 ERROR: Failed to create cache as ROMImageSegment in SH_CacheMap

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative.

JVMSHRC089 ERROR: Cannot create refresh mutex in SH_CacheMap

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low on resources,

please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC090 ERROR: Failed to get cache mutex in SH_CacheMap startup

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low on resources, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC091 ERROR: Read corrupt data for item 0x%p (invalid dataType)

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative.

JVMSHRC092 ERROR: ADD failure when reading cache

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative.

JVMSHRC093 INFO: Detected unexpected termination of another JVM during update

Explanation: The JVM has detected an unexpected termination of another JVM while updating the shared classes cache.

System action: The JVM continues.

User response: No action required, this message is for information only.

JVMSHRC094 ERROR: Orphan found but local ROMClass passed to addROMClassToCache

Explanation: An error has occurred in shared class processing.

System action: The JVM continues if possible.

User response: Contact your IBM service representative.

JVMSHRC095 ERROR: Attempts to call markStale on shared cache items have failed

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative.

JVMSHRC096 WARNING: Shared Cache "%s" is full. Use -Xscmx to set cache size.

Explanation: The named shared classes cache is full and no further classes may be added to it.

System action: The JVM continues. The named shared cache is still operational and continues to provide increased performance for loading the classes it contains. However, classes not contained in the cache will always be loaded from their source.

User response: To gain the full benefit of shared classes, delete the named cache and recreate it specifying a larger shared classes cache size by the -Xscmx parameter.

JVMSHRC097 ERROR: Shared Cache "%s" is corrupt. No new JVMs will be allowed to connect to the cache. Existing JVMs can continue to function, but cannot update the cache.

Explanation: The shared classes cache named in the message is corrupt.

System action: The JVM continues.

User response: Destroy the shared classes cache named in the message and rerun. If the situation persists, contact your IBM service representative.

JVMSHRC125 ERROR: Could not allocate memory for string buffer in SH_CacheMap

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have

specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low of memory resource, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC126 ERROR: Request made to add too many items to ClasspathItem

Explanation: An error has occurred in shared class processing.

System action: The JVM continues.

User response: Contact your IBM service representative.

JVMSHRC127 SH_CompositeCache::enterMutex failed with return code %d

Explanation: An error has occurred while trying to update the shared classes cache.

System action: The JVM will terminate.

User response: The system may be low on resources, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC128 SH_CompositeCache::exitMutex failed with return code %d

Explanation: An error has occurred while trying to update the shared classes cache.

System action: The JVM will terminate.

User response: Contact your IBM service representative.

JVMSHRC129 ERROR: Attempt to set readerCount to -1!

Explanation: An error has occurred while trying to update the shared classes cache.

System action: The JVM continues.

User response: Contact your IBM service representative.

JVMSHRC130 ERROR: Attempt to allocate while commit is still pending

Explanation: An error has occurred while updating the shared classes cache.

System action: The processing will continue, if possible.

User response: Contact your IBM service representative.

JVMSHRC131 ERROR: Cannot allocate memory for linked list item in ROMClassManagerImpl

Explanation: An error has occurred in shared class processing.

System action: The JVM continues.

User response: The system may be low of memory resource, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC132 ERROR: Cannot allocate memory for hashtable entry in ROMClassManagerImpl

Explanation: An error has occurred in shared class processing.

System action: The JVM continues.

User response: The system may be low of memory resource, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC133 ERROR: Cannot enter ROMClassManager hashtable mutex

Explanation: An error has occurred in shared class processing.

System action: The JVM continues.

User response: The system may be low on resources, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC134 ERROR: Failed to create pool in SH_ROMClassManagerImpl

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low on resources, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC135 ERROR: Failed to create hashtable in SH_ROMClassManagerImpl

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with

"-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low on resources, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC136 ERROR: Cannot create monitor in SH_ROMClassManagerImpl

Explanation: An error has occurred in shared class processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: The system may be low on resources, please retry when the system is more lightly loaded. If the situation persists, contact your IBM service representative.

JVMSHRC137 SAFE MODE: Warning: ROMClass %.*s does not match ROMClass in cache

Explanation: This message is issued when running shared classes in safe mode and a mismatch in ROMClass bytes is detected. This message will be followed by further details of the class sizes and details of the mismatched bytes.

System action: The JVM continues.

User response: None required. This message is for information only. The mismatch in bytes does not mean that an error has occurred, but could indicate, for example, that the class has changed since originally stored in the cache.

JVMSHRC147 Character %.*s not valid for cache name

Explanation: The shared classes cache name specified to the JVM contains the indicated invalid character.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Change the requested shared classes cache name so that all characters are valid and rerun.

JVMSHRC154 Escape character %.*s not valid for cache name

Explanation: An invalid escape character has been specified in a shared classes cache name

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM

continues without using Shared Classes.

User response: Specify a valid escape character. Valid escape characters are %u for username (all platforms) and %g for group name (not valid for Windows).

JVMSHRC155 Error copying username into cache name

Explanation: The system has not been able to obtain the username for inclusion in the shared classes cache name.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative.

JVMSHRC156 Error copying groupname into cache name

Explanation: The system has not been able to obtain the groupname for inclusion in the shared classes cache name.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: Contact your IBM service representative.

JVMSHRC157 Unable to allocate %1\$d bytes of shared memory requested Successfully allocated maximum shared memory permitted (%2\$d bytes) (To increase available shared memory, modify system SHMMAX value)

Explanation: The system has not been able to create a shared classes cache of the size requested (1\$). It has been able to create a cache of the maximum size permitted on your system (2\$). This message is specific to Linux systems.

System action: The JVM continues.

User response: If you require a larger cache: destroy this cache, increase the value of SHMMAX and recreate the cache.

JVMSHRC158 Successfully created shared class cache "%1\$s"

Explanation: This message informs you that a shared classes cache with the given name has been created. It is only issued if verbose shared classes messages have been requested with -Xshareclasses:verbose.

System action: The JVM continues.

User response: No action required, this is an information only message.

JVMSHRC159 Successfully opened shared class cache "%1\$s"

Explanation: This message informs you that an existing shared classes cache with the given name has been opened. It is only issued if verbose shared classes messages have been requested with -Xshareclasses:verbose.

System action: The JVM continues.

User response: No action required, this is an information only message.

JVMSHRC160 The wait for the creation mutex while opening semaphore has timed out

Explanation: An error has occurred within Shared Classes processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: If the condition persists, contact your IBM service representative.

JVMSHRC161 The wait for the creation mutex while creating shared memory has timed out

Explanation: An error has occurred within Shared Classes processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: If the condition persists, contact your IBM service representative.

JVMSHRC162 The wait for the creation mutex while opening shared memory has timed out

Explanation: An error has occurred within Shared Classes processing.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: If the condition persists, contact your IBM service representative.

JVMSHRC166 Attached to cache "%1\$s", size=%2\$d bytes

Explanation: This message informs you that you have successfully attached to the cache named 1\$ which is 2\$ bytes in size. This message is only issued if verbose Shared Classes messages have been requested with "-Xshareclasses:verbose".

System action: The JVM continues.

User response: No action required, this is an information only message.

JVMSHRC168 Total shared class bytes read=%1\$lld. Total bytes stored=%2\$d

Explanation: This messages informs you of the number of bytes read from the Shared Classes cache (1\$) and the number of bytes stored in the cache (2\$). It is issued when the JVM exits if you have requested verbose Shared Classes messages with "-Xshareclasses:verbose".

System action: The JVM continues.

User response: No action required, this is an information only message.

**JVMSHRC169 Change detected in %2\$.*1\$s...
...marked %3\$d cached classes stale**

Explanation: This message informs you that a change has been detected in classpath 2\$ and that, as a result, 3\$ classes have been marked as "stale" in the Shared Classes cache. This messages is issued only if you have requested verbose Shared Classes messages with "-Xshareclasses:verbose".

System action: The JVM continues.

User response: No action required, this is an information only message.

JVMSHRC171 z/OS cannot create cache of requested size: Please check your z/OS system BPXPRMxx settings

Explanation: z/OS cannot create a Shared Classes cache of the requested size.

System action: The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.

User response: If you require a cache of this size, ask your system programmer to increase the z/OS system BPXPRMxx settings IPCSHMMPAGES and MAXSHAREPAGES appropriately.

Appendix D. 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 440
- “System property command-line options” on page 442
- “JVM command-line options” on page 444
- “-XX command-line options” on page 452
- “JIT command-line options” on page 452
- “Garbage Collector command-line options” on page 453

Specifying command-line options

Although the command line is the traditional way to specify command-line options, you can pass options to the JVM in other ways.

Use only single or double quotation marks for command-line options when explicitly directed to do so for the option in question. Single and double quotation marks have different meanings on different platforms, operating systems, and shells. Do not use **'-X<option>'** or **"-X<option>"**. Instead, you must use **-X<option>**. For example, do not use **'-Xmx500m'** and **"-Xmx500m"**. Write this option as **-Xmx500m**.

These precedence rules (in descending order) apply to specifying options:

1. Command line.
For example, `java -X<option> MyClass`
2. (i5/OS only) Command-line options can be specified using a `SystemDefault.properties` file. See “Setting default Java command-line options” on page 172 for more information.
3. A file containing a list of options, specified using the **-Xoptionsfile** option on the command line. For example, `java -Xoptionsfile=myoptionfile.txt MyClass`

In the options file, specify each option on a new line; you can use the **'\'** character as a continuation character if you want a single option to span multiple lines. Use the **#** character to define comment lines. You cannot specify **-classpath** in an options file. Here is an example of an options file:

```
#My options file
-X<option1>
-X<option2>=\
<value1>,\
<value2>
-D<sysprop1>=<value1>
```

4. **IBM_JAVA_OPTIONS** environment variable. You can set command-line options using this environment variable. The options that you specify with this environment variable are added to the command line when a JVM starts in that environment.

For example, set `IBM_JAVA_OPTIONS=-X<option1> -X<option2>=<value1>`

General command-line options

Use these options to print help on assert-related options, set the search path for application classes and resources, print a usage method, identify memory leaks inside the JVM, print the product version and continue, enable verbose output, and print the product version.

-assert

Prints help on assert-related options.

-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.

-memorycheck[:<option>]

Identifies memory leaks inside the JVM using strict checks that cause the JVM to exit on failure. If no option is specified, **all** is used by default. Options are:

- **all**
 - The default if **-memorycheck** only is used. This option enables checking of all allocated and freed blocks on every free and allocate call. This check of the heap is the most thorough. It causes the JVM to exit on nearly all memory-related problems soon after they are caused. This option has the greatest affect on performance.
- **callsite=<number of allocations>**
 - Prints callsite information every *<number of allocations>*. Deallocations are not counted. Callsite information is presented in a table with separate information for each callsite. Statistics include the number and size of allocation and free requests since the last report, and the number of the allocation request responsible for the largest allocation from each site. Callsites are presented as `sourcefile:linenumber` for C code and assembly function name for assembler code.

Callsites that do not provide callsite information are accumulated into an “unknown” entry.
- **failat=<number of allocations>**
 - Causes memory allocation to fail (return NULL) after *<number of allocations>*. Setting *<number of allocations>* to 13 causes the 14th allocation to return NULL. Deallocations are not counted. Use this option to ensure that JVM code reliably handles allocation failures. This option is useful for checking allocation site behavior rather than setting a specific allocation limit.
- **nofree**
 - Keeps a list of already used blocks instead of freeing memory. This list is checked, along with currently allocated blocks, 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.

- **quick**

- Enables block padding only. Used to detect basic heap corruption. Pads every allocated block with sentinel bytes, which are verified on every allocate and free. Block padding is faster than the default of checking every block, but is not as effective.

- **skipto=<number of allocations>**

- Causes the program to check only on allocations that occur after <number of allocations>. Deallocations are not counted. Used to speed up JVM startup when early allocations are not causing the memory problem. As a rough estimate, the JVM performs 250+ allocations during startup.

- **zero**

- Newly allocated blocks are set 0 instead of being filled with the 0xE7E7xxxxxxx0E7E7 pattern. Setting to 0 helps you to determine whether a callsite is expecting zeroed memory (in which case after the allocation request by using the instruction `memset(pointer, 0, size)`).

-showversion

Prints product version and continues.

-verbose:<option>[,<option>...]

Enables verbose output. Separate multiple options using commas. These options are available:

class

Writes an entry to stderr for each class that is loaded.

dynload

Provides detailed information as each bootstrap class is loaded by the JVM:

- The class name and package
- For class files that were in a .jar file, the name and directory path of the .jar
- Details of the size of the class and the time taken to load the class

The data is written out to stderr. An example of the output on a Windows platform follows:

```
<Loaded java/lang/String from C:\sdk\jre\lib\vm.jar>
<Class size 17258; ROM size 21080; debug size 0>
<Read time 27368 usec; Load time 782 usec; Translate time 927 usec>
```

gc Provide verbose garbage collection information.

init

Writes information to stderr describing JVM initialization and termination.

jni

Writes information to stderr describing the JNI services called by the application and JVM.

sizes

Writes information to stderr describing the active memory usage settings.

stack

Writes information to stderr describing the Java and C stack usage for each thread.

-version
Prints product version.

System property command-line options

Use the system property command-line options to set up your system.

-D<name>=<value>
Sets a system property.

-DCLONE_HASHTABLE_FOR_SYNCHRONIZATION

Deadlocks can occur when serializing multiple `java.util.Hashtables` that refer to each other in different threads at the same time. Using this command-line option can resolve the deadlock, by forcing the JVM to take a copy of every `java.util.Hashtable` before this hashtable is serialized. Because this process requires temporary storage, and uses additional processing power, the option is not enabled by default.

-Dcom.ibm.jsse2.renegotiate=[ALL | NONE | ABBREVIATED]

If your Java application uses JSSE for secure communication, you can disable TLS renegotiation by installing APAR IZ65239.

ALL Allow both abbreviated and unabbreviated (full) renegotiation handshakes.

NONE Allow no renegotiation handshakes. This value is the default setting.

ABBREVIATED Allow only abbreviated renegotiation handshakes.

-Dcom.ibm.lang.management.verbose

Enables verbose information from `java.lang.management` operations to be written to output channel during VM operation.

-Dcom.ibm.IgnoreMalformedInput=true

From Java 5 SR12, any invalid UTF8 or malformed byte sequences are replaced with the standard unicode replacement character `\uFFFD`. To retain the old behavior, where invalid UTF8 or malformed byte sequences are ignored, set this system property to true.

-Dcom.ibm.mappedByteBufferForce=[true | false]

During system failure, the `MappedByteBuffer.force` API does not commit data to disk, which prevents data integrity issues. Setting this value to true forces data to be committed to disk during system failure. Because this setting can cause performance degradation, this switch is disabled by default.

-Dcom.ibm.nio.DirectByteBuffer.AggressiveMemoryManagement=true

Use this property to increase dynamically the native memory limit for Direct Byte Buffers, based on their usage. This option is applicable when a Java application uses many Direct Byte Buffer objects, but cannot predict the maximum native memory consumption of the objects. Do not use the **-Xsun.nio.MaxDirectMemorySize** option with this property.

-Dcom.ibm.nio.useIBMAlias=true

The IBM JVM cannot display all the Big5-HKSCS characters when using the NIO converter. By specifying the **-Dcom.ibm.nio.useIBMAlias=true** option, you can use the ICU4J API to display Big5-HKSCS characters without modifying the application.

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

Enable the Attach API for this application. The Attach API allows your

application to connect to a virtual machine. Your application can then load an agent application into the virtual machine. The agent can be used to perform tasks such as monitoring the virtual machine status.

-Dcom.ibm.zipfile.closeinputstreams=true

The `Java.util.zip.ZipFile` class allows you to create `InputStreams` on files held in a compressed archive. Under some conditions, using `ZipFile.close()` to close all `InputStreams` that have been opened on the compressed archive might result in a 56-byte-per-`InputStream` native memory leak. Setting the

-Dcom.ibm.zipfile.closeinputstreams=true forces the JVM to track and close `InputStreams` without the memory impact caused by retaining native-backed objects. Native-backed objects are objects that are stored in native memory, rather than the Java heap. By default, the value of this system property is **false**.

-Dibm.jvm.bootclasspath

The value of this property is used as an additional search path, which is inserted between any value that is defined by **-Xbootclasspath/p:** and the bootclass path. The bootclass path is either the default or the one that you defined by using the **-Xbootclasspath:** option.

-Dibm.stream.nio=[true | false]

From v1.4.1 onwards, by default the IO converters are used. This option addresses the ordering of IO and NIO converters. When this option is set to true, the NIO converters are used instead of the IO converters.

-Djava.compiler=[NONE | j9jit23]

Disables the Java compiler by setting to **NONE**. Enable JIT compilation by setting to **j9jit23** (Equivalent to **-Xjit**).

-Djava.net.connectiontimeout=[n]

'n' is the number of seconds to wait for the connection to be established with the server. If this option is not specified in the command line, the default value of 0 (infinity) is used. The value can be used as a timeout limit when an asynchronous java-net application is trying to establish a connection with its server. If this value is not set, a java-net application waits until the default connection timeout value is met. For instance, java **-Djava.net.connectiontimeout=2** `TestConnect` causes the java-net client application to wait only 2 seconds to establish a connection with its server.

-Dsun.awt.keepWorkingSetOnMinimize=true

When a Java application using the Abstract Windowing Toolkit (AWT) is minimized, the default behavior is to “trim” the “working set”. The working set is the application memory stored in RAM. Trimming means that the working set is marked as being available for swapping out if the memory is required by another application. The advantage of trimming is that memory is available for other applications. The disadvantage is that a “trimmed” application might experience a delay as the working set memory is brought back into RAM.

The **-Dsun.awt.keepWorkingSetOnMinimize=true** system property stops the JVM trimming an application when it is minimized. The default behavior is to trim an application when it is minimized.

-Dsun.net.client.defaultConnectTimeout=<value in milliseconds>

Specifies the default value for the connect timeout for the protocol handlers used by the `java.net.URLConnection` class. The default value set by the protocol handlers is -1, which means that no timeout is set.

When a connection is made by an applet to a server and the server does not respond properly, the applet might seem to hang. The delay might also cause the browser to hang. The apparent hang occurs because there is no network connection timeout. To avoid this problem, the Java Plug-in has added a default value to the network timeout of 2 minutes for all HTTP connections. You can override the default by setting this property.

-Dsun.net.client.defaultReadTimeout=<value in milliseconds>

Specifies the default value for the read timeout for the protocol handlers used by the `java.net.URLConnection` class when reading from an input stream when a connection is established to a resource. The default value set by the protocol handlers is -1, which means that no timeout is set.

-Dsun.nio.MaxDirectMemorySize=<value in bytes>

Limits the native memory size for `nio` Direct Byte Buffer objects to the value specified.

-Dsun.rmi.transport.tcp.connectionPool=[true | any non-null value]

Enables thread pooling for the RMI ConnectionHandlers in the TCP transport layer implementation.

-Dsun.timezone.ids.oldermapping=[true | false]

From v5.0 Service Refresh 1 onwards, the Java Virtual Machine uses new time zone identifiers. The identifiers change the definitions of Eastern Standard Time (EST) and Mountain Standard Time (MST). These new definitions do not take daylight saving time (DST) into account. If this property is set to true, the definitions for EST and MST revert to the definitions that were used before v5.0 Service Refresh 1, and DST is taken into account. By default, this property is set to true.

-Dswing.useSystemFontSettings=[false]

From v1.4.1 onwards, by default, Swing programs running with the Windows Look and Feel render with the system font set by the user instead of a Java-defined font. As a result, fonts for v1.4.1 differ from the fonts in earlier releases. This option addresses compatibility problems like these for programs that depend on the old behavior. By setting this option, v1.4.1 fonts and those of earlier releases are the same for Swing programs running with the Windows Look and Feel.

JVM command-line options

Use these options to configure your JVM. The options prefixed with **-X** are nonstandard.

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

For options that take a *<percentage>* parameter, use a number from 0 to 1. For example, 50% is 0.5.

Options that relate to the JIT are listed under "JIT command-line options" on page 452. Options that relate to the Garbage Collector are listed under "Garbage Collector command-line options" on page 453.

-X Displays help on nonstandard options.

-Xaggressive

(AIX 32-bit, Linux PPC32 and Windows 64-bit only) Enable performance optimizations that are expected to be the default in future releases.

-Xargencoding

You can put Unicode escape sequences in the argument list. This option is set to off by default.

-Xbootclasspath:<directories and compressed or Java archive files separated by : (; on Windows)>

Sets the search path for bootstrap classes and resources. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xbootclasspath/a:<directories and compressed or Java archive files separated by : (; on Windows)>

Appends the specified directories, compressed files, or jar files to the end of the bootstrap class path. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xbootclasspath/p:<directories and compressed or Java archive files separated by : (; on Windows)>

Adds a prefix of the specified directories, compressed files, or Java archive files to the front of the bootstrap class path. Do not deploy applications that use the **-Xbootclasspath:** or the **-Xbootclasspath/p:** option to override a class in the standard API. This is because such a deployment contravenes the Java 2 Runtime Environment binary code license. The default is to search for bootstrap classes and resources in the internal VM directories and .jar files.

-Xcheck:jni[:<option>=<value>]

Performs additional checks for JNI functions. This option is equivalent to **-Xrunjchk**. By default, no checking is performed.

-Xclassgc

Enables dynamic unloading of classes by the JVM. This is the default behavior. To disable dynamic class unloading, use the **-Xnoclassgc** option.

-Xdbg:<options>

Loads debugging libraries to support the remote debugging of applications. This option is equivalent to **-Xrunjdwp**. By default, the debugging libraries are not loaded, and the VM instance is not enabled for debug.

-Xdbginfo:<path to symbol file>

Loads and passes options to the debug information server. By default, the debug information server is disabled.

-Xdebug

This option is deprecated. Use **-Xdbg** for debugging.

-Xdiagnosticscollector[:settings=<filename>]

Enables the Diagnostics Collector. See Chapter 27, “The Diagnostics Collector,” on page 323 for more information. The settings option allows you to specify a different Diagnostics Collector settings file to use instead of the default dc.properties file in the JRE.

-Xdisablejavadump

Turns off Javacore generation on errors and signals. By default, Javacore generation is enabled.

-Xdump

See Chapter 21, “Using dump agents,” on page 223.

-Xenableexplicitgc

Signals to the VM that calls to `System.gc()` trigger a garbage collection. This option is enabled by default.

-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.

-Xifa:<on | off | force> (z/OS only)

z/OS R6 can run Java applications on a new type of special-purpose assist processor called the *eServer™ zSeries Application Assist Processor (zAAP)*. The zSeries Application Assist Processor is also known as an IFA (Integrated Facility for Applications).

The **-Xifa** option enables Java applications to run on IFAs if they are available. The default value for the **-Xifa** option is *on*. Only Java code and system native methods can be on IFA processors.

-Xiss<size>

Sets the initial stack size for Java threads. By default, the stack size is set to 2 KB. Use the **-verbose:sizes** option to output the value that the VM is using.

-Xjarversion

Produces output information about the version of each jar file in the class path, the boot class path, and the extensions directory. Version information is taken from the Implementation-Version and Build-Level properties in the manifest of the jar.

-Xjni:<suboptions>

Sets JNI options. You can use the following suboption with the **-Xjni** option:

-Xjni:arrayCacheMax=[<size in bytes> | unlimited]

Sets the maximum size of the array cache. The default size is 8096 bytes.

-Xlinenumbers

Displays line numbers in stack traces for debugging. See also **-Xnolinenumbers**. By default, line numbers are on.

-XlockReservation

Enables an optimization that presumes a monitor is owned by the thread that last acquired it. The optimization minimizes the runtime cost of acquiring and releasing a monitor for a single thread if the monitor is rarely acquired by multiple threads.

-Xlog

Enables message logging. To prevent message logging, use the **-Xlog:none** option. By default, logging is enabled. This option is available from Java 5 SR10. See Appendix C, "Messages," on page 419.

-Xlp<size> (AIX, Windows, and Linux (x86, PPC32, PPC64, AMD64, EM64T))

AIX: Requests the JVM to allocate the Java heap (the heap from which Java objects are allocated) with large (16 MB) pages, if a size is not specified. If large pages are not available, the Java heap is allocated with the next smaller page size that is supported by the system. AIX requires special configuration to enable large pages. For more information about configuring AIX support for large pages, see http://www.ibm.com/servers/aix/whitepapers/large_page.html. The SDK supports the use of large pages only to back the Java heap shared memory segments. The JVM uses `shmget()` with the

SHM_LGPG and SHM_PIN flags to allocate large pages. The **-Xlp** option replaces the environment variable **IBM_JAVA_LARGE_PAGE_SIZE**, which is now ignored if set.

Linux: Requests the JVM to allocate the Java heap with large pages. If large pages are not available, the JVM does not start, displaying the error message GC: system configuration does not support option --> '-Xlp'. The JVM uses shmget() to allocate large pages for the heap. Large pages are supported by systems running Linux kernels v2.6 or higher. By default, large pages are not used.

Windows: Requests the JVM to allocate the Java heap with large pages. This command is available only on Windows Server 2003, Windows Vista, and Windows Server 2008. To use large pages, the user that runs Java must have the authority to “lock pages in memory”. To enable this authority, as Administrator go to **Control Panel** → **Administrative Tools** → **Local Security Policy** and then find **Local Policies** → **User Rights Assignment** → **Lock pages in memory**. Add the user who runs the Java process, and reboot your machine. For more information, see these Web sites:

- <http://technet2.microsoft.com/WindowsServer/en/library/e72dcdf6-fe17-49dd-a382-02baad31a1371033.msp>
- [http://msdn.microsoft.com/en-us/library/aa366720\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa366720(VS.85).aspx)
- [http://msdn.microsoft.com/en-us/library/aa366568\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa366568(VS.85).aspx)

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

z/OS: Large pages are not supported on z/OS for Java 5.

-Xmso<size>

Sets the initial stack size for operating system threads. The default value can be determined by running the command:

```
java -verbose:sizes
```

The maximum value for the stack size varies according to platform and specific machine configuration. If you exceed the maximum value, a java/lang/OutOfMemoryError message is reported.

-Xmxcl<number>

Sets the maximum number of class loaders. See “OutOfMemoryError exception when using delegated class loaders” on page 30 for a description of a problem that can occur on some JVMs if this number is set too low.

-Xnoagent

Disables support for the old JDB debugger.

-Xnoclassgc

Disables dynamic class unloading. This option disables the release of native and Java heap storage associated with Java class loaders and classes that are no longer being used by the JVM. The default behavior is as defined by

-Xclassgc. Enabling this option is not recommended except under the direction

of the IBM Java support team. The reason is the option can cause unlimited native memory growth, leading to out-of-memory errors.

-Xnolinenumbers

Disables the line numbers for debugging. See also **-Xlinenumbers**. By default, line number are on.

-Xnosigcatch

Disables JVM signal handling code. See also **-Xsigcatch**. By default, signal handling is enabled.

-Xnosigchain

Disables signal handler chaining. See also **-Xsigchain**. By default, the signal handler chaining is enabled, except for z/OS.

-Xoptionsfile=<file>

Specifies a file that contains JVM options and definitions. By default, no option file is used.

The options file does not support these options:

- **-assert**
- **-fullversion**
- **-help**
- **-memorycheck**
- **-showversion**
- **-version**
- **-Xjarversion**
- **-Xoptionsfile**

Although you cannot use **-Xoptionsfile** recursively within an options file, you can use **-Xoptionsfile** multiple times on the same command line to load more than one options files.

<file> contains options that are processed as if they had been entered directly as command-line options. For example, the options file might contain:

```
-DuserString=ABC123
-Xmx256MB
```

Some options use quoted strings as parameters. Do not split quoted strings over multiple lines using the line continuation character '\'. The '¥' character is not supported as a line continuation character. For example, the following example is not valid in an options file:

```
-Xevents=vmstop,exec="cmd /c \
echo %pid has finished."
```

The following example is valid in an options file:

```
-Xevents=vmstop, \
exec="cmd /c echo %pid has finished."
```

-Xoss<size>

Recognized but deprecated. Use **-Xss** and **-Xms0**. Sets the maximum Java stack size for any thread. The maximum value for the stack size varies according to platform and specific machine configuration. If you exceed the maximum value, a java/lang/OutOfMemoryError message is reported.

-Xrdbginfo:<host>:<port>

Loads the remote debug information server with the specified host and port. By default, the remote debug information server is disabled.

-Xrs

Disables signal handling in the JVM. Setting **-Xrs** prevents the Java runtime from handling any internally or externally generated signals such as SIGSEGV and SIGABRT. Any signals raised are handled by the default operating system handlers. Disabling signal handling in the JVM reduces performance by approximately 2-4%, depending on the application.

Note: Linux always uses SIGUSR1.

-Xrs:sync

On UNIX systems, this option disables signal handling in the JVM for SIGSEGV, SIGFPE, SIGBUS, SIGILL, SIGTRAP, and SIGABRT signals. However, the JVM still handles the SIGQUIT and SIGTERM signals, among others. On Windows systems, hardware exceptions are not handled by the JVM when this option is specified. However, the Windows CTRL_BREAK_EVENT signal, triggered by the Ctrl-Break key combination, is still handled by the JVM. As with **-Xrs**, the use of **-Xrs:sync** reduces performance by approximately 2-4%, depending on the application.

-Xrun<library name>[:<options>]

Loads helper libraries. To load multiple libraries, specify it more than once on the command line. Examples of these libraries are:

-Xrunhprof[:help] | [[:<option>=<value>, ...]

Performs heap, CPU, or monitor profiling.

-Xrunjdwp[:help] | [[:<option>=<value>, ...]

Loads debugging libraries to support the remote debugging of applications. This option is the same as **-Xdbg**.

-Xrunjnic[k[:help] | [[:<option>=<value>, ...]

Deprecated. Use **-Xcheck:jni** instead.

-Xscmx<size>

Specifies cache size. This option applies only if a cache is being created and no cache of the same name exists. The default cache size is platform-dependent. You can find out the size value being used by adding **-verbose:sizes** as a command-line argument. Minimum cache size is 4 KB. Maximum cache size is platform-dependent. The size of cache that you can specify is limited by the amount of physical memory and paging space available to the system. The virtual address space of a process is shared between the shared classes cache and the Java heap. Increasing the maximum size of the Java heap reduces the size of the shared classes cache that you can create.

-Xshareclasses:<suboptions>

Enables class sharing. This option can take a number of suboptions, some of which are cache utilities. Cache utilities perform the required operation on the specified cache, without starting the VM. You can combine multiple suboptions, separated by commas, but the cache utilities are mutually exclusive.

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

destroy (Utility option)

Destroys a cache using the name specified in the **name=<name>** suboption. If the name is not specified, the default cache is destroyed. A cache can be destroyed only if all virtual machines using it have shut down, and the user has sufficient permissions.

destroyAll (Utility option)

Tries to destroy all caches available to the user. A cache can be destroyed only if all virtual machines using it have shut down, and the user has sufficient permissions.

expire=<time in minutes> (Utility option)

Destroys all caches that have been unused for the time specified before loading shared classes. This option is not a utility option because it does not cause the JVM to exit. On NTFS file systems, the expire option is accurate to the nearest hour.

groupAccess

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

help

Lists all the command-line options.

listAllCaches (Utility option)

Lists all the caches on the system, describing if they are in use and when they were last used.

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 356 for more information.

name=<name>

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

none

Added to the end of a command line, disables class data sharing. This suboption overrides class sharing arguments found earlier on the command line.

nonfatal

Allows the JVM to start even if class data sharing fails. Normal behavior for the JVM is to refuse to start if class data sharing fails. If you select **nonfatal** and the shared classes cache fails to initialize, the JVM starts without class data sharing.

printAllStats (Utility option)

Displays detailed information about the contents of the cache specified in the **name=<name>** suboption. If the name is not specified, statistics are displayed about the default cache. Every class is listed in chronological order with a reference to the location from which it was loaded. See “printAllStats utility” on page 365 for more information.

printStats (Utility option)

Displays summary statistics information about the cache specified in the **name=<name>** suboption. If the name is not specified, statistics are displayed about the default cache. The most useful information displayed is how full the cache is and how many classes it contains. Stale classes are classes that have been updated on the file system and which the cache has therefore marked "stale". Stale classes are not purged from the cache. See "printStats utility" on page 364 for more information.

safemode

Forces the JVM to load all classes from disk and apply the modifications to those classes (if applicable). See "Using the safemode option" on page 357 for more information.

silent

Disables all shared class messages, including error messages. Unrecoverable error messages, which prevent the JVM from initializing, are displayed.

verbose

Gives detailed output on the cache I/O activity, listing information about classes being stored and found. Each class loader is given a unique ID (the bootstrap loader is always 0) and the output shows the class loader hierarchy at work, where class loaders must ask their parents for a class before they can load it themselves. It is typical to see many failed requests; this behavior is expected for the class loader hierarchy. The standard option **-verbose:class** also enables class sharing verbose output if class sharing is enabled.

verboseHelper

Enables verbose output for the Java Helper API. This output shows you how the Helper API is used by your ClassLoader.

verboseIO

Gives detailed output on the cache I/O activity, listing information about classes being stored and found. Each class loader is given a unique ID (the bootstrap loader is always 0) and the output shows the class loader hierarchy at work, where class loaders must ask their parents for a class before they can load it themselves. It is typical to see many failed requests; this behavior is expected for the class loader hierarchy.

-Xsigcatch

Enables VM signal handling code. See also **-Xnosigcatch**. By default, signal handling is enabled.

-Xsigchain

Enables signal handler chaining. See also **-Xnosigchain**. By default, signal handler chaining is enabled.

-Xss<size>

Sets the maximum stack size for Java threads. The default is 256 KB for 32-bit JVMs and 512 KB for 64-bit JVMs. The maximum value varies according to platform and specific machine configuration. If you exceed the maximum value, a `java/lang/OutOfMemoryError` message is reported.

-Xssi<size>

Sets the stack size increment for Java threads. When the stack for a Java thread becomes full it is increased in size by this value until the maximum size (**-Xss**) is reached. The default is 16 KB.

-Xthr:minimizeUserCPU

Minimizes user-mode CPU usage in thread synchronization where possible. The reduction in CPU usage might be a trade-off in exchange for lower performance.

-XtlhPrefetch

Speculatively prefetches bytes in the thread local heap (TLH) ahead of the current allocation pointer during object allocation. This helps reduce the performance cost of subsequent allocations.

-Xtrace[:help] | [[:<option>=<value>, ...]

See “Controlling the trace” on page 287 for more information.

-Xverify[:<option>]

With no parameters, enables the verifier, which is the default. Therefore, if used on its own with no parameters, for example, **-Xverify**, this option does nothing. Optional parameters are as follows:

- **all** - enable maximum verification
- **none** - disable the verifier
- **remote** - enables strict class-loading checks on remotely loaded classes

The verifier is on by default and must be enabled for all production servers. Running with the verifier off is not a supported configuration. If you encounter problems and the verifier was turned off using **-Xverify:none**, remove this option and try to reproduce the problem.

-XX command-line options

JVM command-line options that are specified with -XX are not stable and are not recommended for casual use.

These options are subject to change without notice.

-XXallowvmshutdown[:false|true]

This option is provided as a workaround for customer applications that cannot shut down cleanly, as described in APAR IZ59734. Customers who need this workaround should use **-XXallowvmshutdown:false**. The default option is **-XXallowvmshutdown:true** for Java 5 SR10 onwards.

-XX:-StackTraceInThrowable

This option removes stack traces from exceptions. By default, stack traces are available in exceptions. Including a stack trace in exceptions requires walking the stack and that can affect performance. Removing stack traces from exceptions can improve performance but can also make problems harder to debug.

When this option is enabled, `Throwable.getStackTrace()` returns an empty array and the stack trace is displayed when an uncaught exception occurs. `Thread.getStackTrace()` and `Thread.getAllStackTraces()` are not affected by this option.

JIT command-line options

Use these JIT compiler command-line options to control code compilation.

For more information about JIT, see Chapter 26, “JIT problem determination,” on page 317.

-Xcodecache<size>

This option is used to tune performance. It sets the size of each block of

memory that is allocated to store the native code of compiled Java methods. By default, this size is selected internally according to the processor architecture and the capability of your system. If profiling tools show significant costs in trampolines, that is a good reason to change the size until the costs are reduced. Changing the size does not mean always increasing the size. The option provides the mechanism to tune for the right size until hot interblock calls are eliminated. A reasonable starting point to tune for the optimal size is (totalNumberByteOfCompiledMethods * 1.1).

Note: Trampolines are where reflection is used to avoid inner classes. JVMTI identifies trampolines in a methodLoad2 event.

-Xcomp (z/OS only)

Forces methods to be compiled by the JIT compiler on their first use. The use of this option is deprecated; use **-Xjit:count=0** instead.

-Xint

Makes the JVM use the Interpreter only, disabling the Just-In-Time (JIT) compilers. By default, the JIT compiler is enabled.

-Xjit[:<parameter>=<value>, ...]

With no parameters, enables the JIT compiler. The JIT compiler is enabled by default, so using this option on its own has no effect. Use this option to control the behavior of the JIT compiler. Useful parameters are:

count=<n>

Where <n> is the number of times a method is called before it is compiled. For example, setting count=0 forces the JIT compiler to compile everything on first execution.

limitFile=(<filename>, <m>, <n>)

Compile only the methods listed on lines <m> to <n> in the specified limit file. Methods not listed in the limit file and methods listed on lines outside the range are not compiled.

optlevel=[noOpt | cold | warm | hot | veryHot | scorching]

Forces the JIT compiler to compile all methods at a specific optimization level. Specifying **optlevel** might have an unexpected effect on performance, including lower overall performance.

verbose

Reports information about the JIT and AOT compiler configuration and method compilation.

-Xquickstart

Causes the JIT compiler to run with a subset of optimizations. The effect is faster compilation times that improve startup time, but longer running applications might run slower. **-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.

-XsamplingExpirationTime<time> (from Service Refresh 5)

Disables the JIT sampling thread after <time> seconds. When the JIT sampling thread is disabled, no processor cycles are used by an idle JVM.

Garbage Collector command-line options

Use these Garbage Collector command-line options to control garbage collection.

You might need to read Chapter 2, “Memory management,” on page 7 to understand some of the references that are given here.

The **-verbose:gc** option detailed in “-verbose:gc logging” on page 330 is the main diagnostic aid that is available for runtime analysis of the Garbage Collector. However, additional command-line options are available that affect the behavior of the Garbage Collector and might aid diagnostics.

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

For options that take a *<percentage>* parameter, use a number from 0 to 1, for example, 50% is 0.5.

-Xalwaysclassgc

Always perform dynamic class unloading checks during global collection. The default behavior is as defined by **-Xclassgc**.

-Xclassgc

Enables the collection of class objects only on class loader changes. This behavior is the default.

-Xcompactexplicitgc

Enables full compaction each time `System.gc()` is called.

-Xcompactgc

Compacts on all garbage collections (system and global).

The default (no compaction option specified) makes the GC compact based on a series of triggers that attempt to compact only when it is beneficial to the future performance of the JVM.

-Xconcurrentbackground<number>

Specifies the number of low-priority background threads attached to assist the mutator threads in concurrent mark. The default is 1.

-Xconcurrentlevel<number>

Specifies the allocation "tax" rate. It indicates the ratio between the amount of heap allocated and the amount of heap marked. The default is 8.

-Xconmeter:<soa | loa | dynamic>

This option determines the usage of which area, LOA (Large Object Area) or SOA (Small Object Area), is metered and hence which allocations are taxed during concurrent mark. Using **-Xconmeter:soa** (the default) applies the allocation tax to allocations from the small object area (SOA). Using **-Xconmeter:loa** applies the allocation tax to allocations from the large object area (LOA). If **-Xconmeter:dynamic** is specified, the collector dynamically determines which area to meter based on which area is exhausted first, whether it is the SOA or the LOA.

-Xdisableexcessivegc

Disables the throwing of an `OutOfMemory` exception if excessive time is spent in the GC.

-Xdisableexplicitgc

Disables `System.gc()` calls.

Many applications still make an excessive number of explicit calls to `System.gc()` to request garbage collection. In many cases, these calls degrade

performance through premature garbage collection and compactions. However, you cannot always remove the calls from the application.

The **-Xdisableexplicitgc** parameter allows the JVM to ignore these garbage collection suggestions. Typically, system administrators use this parameter in applications that show some benefit from its use.

By default, calls to `System.gc()` trigger a garbage collection.

-Xdisablestringconstantgc

Prevents strings in the string intern table from being collected.

-Xenableexcessivegc

If excessive time is spent in the GC, the option returns null for an allocate request and thus causes an `OutOfMemory` exception to be thrown. This action occurs only when the heap has been fully expanded and the time spent is making up at least 95%. This behavior is the default.

-Xenablestringconstantgc

Enables strings from the string intern table to be collected. This behavior is the default.

-Xgc:<options>

Passes options such as `verbose`, `compact`, and `nocompact` to the Garbage Collector.

-Xgcpolicy:<optthruput | optavgpause | gencon | subpool (AIX, Linux and IBM i on IBM POWER® architecture, Linux and z/OS on zSeries) >

Controls the behavior of the Garbage Collector.

The `optthruput` option is the default and delivers high throughput to applications, but at the cost of occasional pauses. Disables concurrent mark.

The `optavgpause` option reduces the time that is spent in these garbage collection pauses and limits the effect of increasing heap size on the length of the garbage collection pause. Use `optavgpause` if your configuration has a large heap. Enables concurrent mark.

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

The `subpool` option (AIX, Linux and IBM i on IBM POWER architecture, and z/OS) uses an improved object allocation algorithm to achieve better performance when allocating objects on the heap. This option might improve performance on large SMP systems.

-Xgcthreads<number>

Sets the number of threads that the Garbage Collector uses for parallel operations. This total number of GC threads is composed of one application thread with the remainder being dedicated GC threads. By default, the number is set to the number of physical CPUs present. To set it to a different number (for example 4), use **-Xgcthreads4**. The minimum valid value is 1, which disables parallel operations, at the cost of performance. No advantage is gained if you increase the number of threads above the default setting; you are recommended not to do so.

On systems running multiple JVMs or in LPAR environments where multiple JVMs can share the same physical CPUs, you might want to restrict the number of GC threads used by each JVM. The restriction helps prevent the total number of parallel operation GC threads for all JVMs exceeding the number of physical CPUs present, when multiple JVMs perform garbage collection at the same time.

-Xgcworkpackets<number>

Specifies the total number of work packets available in the global collector. If not specified, the collector allocates a number of packets based on the maximum heap size.

-Xloa

Allocates a large object area (LOA). Objects are allocated in this LOA rather than the SOA. By default, the LOA is enabled for all GC policies except for subpool, where the LOA is not available.

-Xloainitial<percentage>

Specifies the initial percentage (between 0 and 0.95) of the current tenure space allocated to the large object area (LOA). The default value is 0.05, which is 5%.

-Xloamaximum<percentage>

Specifies the maximum percentage (between 0 and 0.95) of the current tenure space allocated to the large object area (LOA). The default value is 0.5, which is 50%.

-Xmaxe<size>

Sets the maximum amount by which the garbage collector expands the heap. Typically, the garbage collector expands the heap when the amount of free space falls below 30% (or by the amount specified using **-Xminf**), by the amount required to restore the free space to 30%. The **-Xmaxe** option limits the expansion to the specified value; for example **-Xmaxe10M** limits the expansion to 10 MB. By default, there is no maximum expansion size.

-Xmaxf<percentage>

Specifies the maximum percentage of heap that must be free after a garbage collection. If the free space exceeds this amount, the JVM tries to shrink the heap. The default value is 0.6 (60%).

-Xmaxt<percentage>

Specifies the maximum percentage of time to be spent in Garbage Collection. If the percentage of time rises above this value, the JVM tries to expand the heap. The default value is 13%.

-Xmca<size>

Sets the expansion step for the memory allocated to store the RAM portion of loaded classes. Each time more memory is required to store classes in RAM, the allocated memory is increased by this amount. By default, the expansion step is 32 KB. Use the **-verbose:sizes** option to determine the value that the VM is using. If the expansion step size you choose is too large, `OutOfMemoryError` is reported. The exact value of a “too large” expansion step size varies according to the platform and the specific machine configuration.

-Xmco<size>

Sets the expansion step for the memory allocated to store the ROM portion of loaded classes. Each time more memory is required to store classes in ROM, the allocated memory is increased by this amount. By default, the expansion step is 128 KB. Use the **-verbose:sizes** option to determine the value that the VM is using. If the expansion step size you choose is too large, `OutOfMemoryError` is reported. The exact value of a “too large” expansion step size varies according to the platform and the specific machine configuration.

-Xmine<size>

Sets the minimum amount by which the Garbage Collector expands the heap. Typically, the garbage collector expands the heap by the amount required to restore the free space to 30% (or the amount specified using **-Xminf**). The **-Xmine** option sets the expansion to be at least the specified value; for

example, `-Xmine50M` sets the expansion size to a minimum of 50 MB. By default, the minimum expansion size is 1 MB.

-Xminf<percentage>

Specifies the minimum percentage of heap to be left free after a garbage collection. If the free space falls below this amount, the JVM attempts to expand the heap. The default value is 30%.

-Xmint<percentage>

Specifies the minimum percentage of time to spend in Garbage Collection. If the percentage of time drops below this value, the JVM tries to shrink the heap. The default value is 5%.

-Xmn<size>

Sets the initial and maximum size of the new area to the specified value when using `-Xgcpolicy:gencon`. Equivalent to setting both `-Xmns` and `-Xmnx`. If you set either `-Xmns` or `-Xmnx`, you cannot set `-Xmn`. If you try to set `-Xmn` with either `-Xmns` or `-Xmnx`, the VM does not start, returning an error. By default, `-Xmn` is not set. If the scavenger is disabled, this option is ignored.

-Xmns<size>

Sets the initial size of the new area to the specified value when using `-Xgcpolicy:gencon`. By default, this option is set to 25% of the value of the `-Xms` option or 64 MB, whichever is less. This option returns an error if you try to use it with `-Xmn`. You can use the `-verbose:sizes` option to find out the values that the VM is currently using. If the scavenger is disabled, this option is ignored.

-Xmnx<size>

Sets the maximum size of the new area to the specified value when using `-Xgcpolicy:gencon`. By default, this option is set to 25% of the value of the `-Xmx` option or 64 MB, whichever is less. This option returns an error if you try to use it with `-Xmn`. You can use the `-verbose:sizes` option to find out the values that the VM is currently using. If the scavenger is disabled, this option is ignored.

-Xmo<size>

Sets the initial and maximum size of the old (tenured) heap to the specified value when using `-Xgcpolicy:gencon`. Equivalent to setting both `-Xmos` and `-Xmox`. If you set either `-Xmos` or `-Xmox`, you cannot set `-Xmo`. If you try to set `-Xmo` with either `-Xmos` or `-Xmox`, the VM does not start, returning an error. By default, `-Xmo` is not set.

-Xmoi<size>

Sets the amount the Java heap is incremented when using `-Xgcpolicy:gencon`. If set to zero, no expansion is allowed. By default, the increment size is calculated on the expansion size, set by `-Xmine` and `-Xminf`.

-Xmos<size>

Sets the initial size of the old (tenure) heap to the specified value when using `-Xgcpolicy:gencon`. By default, this option is set to the value of the `-Xms` option minus the value of the `-Xmns` option. This option returns an error if you try to use it with `-Xmo`. You can use the `-verbose:sizes` option to find out the values that the VM is currently using.

-Xmox<size>

Sets the maximum size of the old (tenure) heap to the specified value when using `-Xgcpolicy:gencon`. By default, this option is set to the same value as the

-Xmx option. This option returns an error if you try to use it with **-Xmo**. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

-Xmr<size>

Sets the size of the Garbage Collection "remembered set". This set is a list of objects in the old (tenured) heap that have references to objects in the new area. By default, this option is set to 16 K.

-Xmrx<size>

Sets the remembered maximum size setting.

-Xms<size>

Sets the initial Java heap size. You can also use the **-Xmo** option. The minimum size is 8 KB.

If scavenger is enabled, **-Xms** >= **-Xmn** + **-Xmo**.

If scavenger is disabled, **-Xms** >= **-Xmo**.

-Xmx<size>

Sets the maximum memory size (**-Xmx** >= **-Xms**)

Examples of the use of **-Xms** and **-Xmx**:

-Xms2m -Xmx64m

Heap starts at 2 MB and grows to a maximum of 64 MB.

-Xms100m -Xmx100m

Heap starts at 100 MB and never grows.

-Xms20m -Xmx1024m

Heap starts at 20 MB and grows to a maximum of 1 GB.

-Xms50m

Heap starts at 50 MB and grows to the default maximum.

-Xmx256m

Heap starts at default initial value and grows to a maximum of 256 MB.

-Xnocsrgc

Disables class garbage collection. This option switches off garbage collection of storage associated with Java classes that are no longer being used by the JVM. The default behavior is as defined by **-Xclassgc**. By default, class garbage collection is performed.

-Xnocompactexplicitgc

Disables compaction on `System.gc()` calls. Compaction takes place on global garbage collections if you specify **-Xcompactgc** or if compaction triggers are met. By default, compaction is enabled on calls to `System.gc()`.

-Xnocompactgc

Disables compaction on all garbage collections (system or global). By default, compaction is enabled.

-Xnoloa

Prevents allocation of a large object area; all objects are allocated in the SOA. See also **-Xloa**.

-Xnopartialcompactgc

Disables incremental compaction. See also **-Xpartialcompactgc**.

-Xpartialcompactgc

Enables incremental compaction. See also **-Xnopartialcompactgc**. By default, this option is not set, so all compactations are full.

-Xsoftmx<size> (AIX only)

This option sets the initial maximum size of the Java heap. Use the **-Xmx** option to set the maximum heap size. Use the AIX DLPAR API in your application to alter the heap size limit between **-Xms** and **-Xmx** at run time. By default, this option is set to the same value as **-Xmx**.

-Xsoftrefthreshold<number>

Sets the number of GCs after which a soft reference is cleared if its referent has not been marked. The default is 32, meaning that on the 32nd GC where the referent is not marked the soft reference is cleared.

-Xtgc:<arguments>

Provides GC tracing options, where *<arguments>* is a comma-separated list containing one or more of the following arguments:

backtrace

Before a garbage collection, a single line is printed containing the name of the master thread for garbage collection, as well as the value of the `osThread` slot in the `J9VMThread` structure.

compaction

Prints extra information showing the relative time spent by threads in the “move” and “fixup” phases of compaction

concurrent

Prints extra information showing the activity of the concurrent mark background thread

dump

Prints a line of output for every free chunk of memory in the system, including “dark matter” (free chunks that are not on the free list for some reason, typically because they are too small). Each line contains the base address and the size in bytes of the chunk. If the chunk is followed in the heap by an object, the size and class name of the object is also printed. This argument has a similar effect to the **terse** argument.

freeList

Before a garbage collection, prints information about the free list and allocation statistics since the last GC. Prints the number of items on the free list, including “deferred” entries (with the scavenger, the unused space is a deferred free list entry). For TLH and non-TLH allocations, prints the total number of allocations, the average allocation size, and the total number of bytes discarded during allocation. For non-TLH allocations, also included is the average number of entries that were searched before a sufficiently large entry was found.

parallel

Produces statistics on the activity of the parallel threads during the mark and sweep phases of a global GC.

references

Prints extra information every time that a reference object is enqueued for finalization, showing the reference type, reference address, and referent address.

scavenger

Prints extra information after each scavenger collection. A histogram is produced showing the number of instances of each class, and their relative ages, present in the survivor space. The information is obtained by performing a linear walk-through of the space.

terse

Dumps the contents of the entire heap before and after a garbage collection. For each object or free chunk in the heap, a line of trace output is produced. Each line contains the base address, "a" if it is an allocated object, and "f" if it is a free chunk, the size of the chunk in bytes, and, if it is an object, its class name.

-Xverbosegclog[:<file>[,<X>,<Y>]]

Causes **-verbose:gc** output to be written to the specified file. If the file cannot be found, **-verbose:gc** tries to create the file, and then continues as normal if it is successful. If it cannot create the file (for example, if an invalid filename is passed into the command), it redirects the output to stderr.

If you specify <X> and <Y> the **-verbose:gc** output is redirected to X files, each containing Y GC cycles.

The dump agent tokens can be used in the filename. See "Dump agent tokens" on page 238 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.

Appendix E. Default settings for the JVM

This appendix shows the default settings that the JVM uses. These settings affect how the JVM operates if you do not apply any changes to its environment. The tables show the JVM operation and the default setting.

These tables are a quick reference to the state of the JVM when it is first installed. The last column shows how the default setting can be changed:

- c** The setting is controlled by a command-line parameter only.
- e** The setting is controlled by an environment variable only.
- ec** The setting is controlled by a command-line parameter or an environment variable. The command-line parameter always takes precedence.

JVM setting	Default	Setting affected by
Javadumps	Enabled	ec
Javadumps on out of memory	Enabled	ec
Heapdumps	Disabled	ec
Heapdumps on out of memory	Enabled	ec
Sysdumps	Enabled	ec
Where dump files are produced	Current [®] directory	ec
Verbose output	Disabled	c
Boot classpath search	Disabled	c
JNI checks	Disabled	c
Remote debugging	Disabled	c
Strict conformance checks	Disabled	c
Quickstart	Disabled	c
Remote debug info server	Disabled	c
Reduced signaling	Disabled	c
Signal handler chaining	Enabled	c
Classpath	Not set	ec
Class data sharing	Disabled	c
Accessibility support	Enabled	e
JIT compiler	Enabled	ec
JIT debug options	Disabled	c
Java2D max size of fonts with algorithmic bold	14 point	e
Java2D use rendered bitmaps in scalable fonts	Enabled	e
Java2D freetype font rasterizing	Enabled	e
Java2D use AWT fonts	Disabled	e

JVM setting	AIX	IBM i	Linux	Windows	z/OS	Setting affected by
Default locale	None	None	None	N/A	None	e
Time to wait before starting plug-in	N/A	N/A	Zero	N/A	N/A	e
Temporary directory	/tmp	/tmp	/tmp	c:\temp	/tmp	e
Plug-in redirection	None	None	None	N/A	None	e
IM switching	Disabled	Disabled	Disabled	N/A	Disabled	e
IM modifiers	Disabled	Disabled	Disabled	N/A	Disabled	e
Thread model	N/A	N/A	N/A	N/A	Native	e
Initial stack size for Java Threads 32-bit. Use: -Xiss<size>	2 KB	2 KB	2 KB	2 KB	2 KB	c
Maximum stack size for Java Threads 32-bit. Use: -Xss<size>	256 KB	256 KB	256 KB	256 KB	256 KB	c
Stack size for OS Threads 32-bit. Use -Xmso<size>	256 KB	256 KB	256 KB	32 KB	256 KB	c
Initial stack size for Java Threads 64-bit. Use: -Xiss<size>	2 KB	N/A	2 KB	2 KB	2 KB	c
Maximum stack size for Java Threads 64-bit. Use: -Xss<size>	512 KB	N/A	512 KB	512 KB	512 KB	c
Stack size for OS Threads 64-bit. Use -Xmso<size>	256 KB	N/A	256 KB	256 KB	256 KB	c
Initial heap size. Use -Xms<size>	4 MB	4 MB	4 MB	4 MB	1 MB	c
Maximum Java heap size. Use -Xmx<size>	64 MB	2 GB	Half the real memory with a minimum of 16 MB and a maximum of 512 MB	Half the real memory with a minimum of 16 MB and a maximum of 2 GB	64 MB	c

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM United Kingdom Laboratories, MP146, Hursley Park, Winchester, Hampshire, SO21 2JN, United Kingdom. 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 information 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.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. 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 IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols

indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at <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 are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product and service names may be trademarks or service marks of others.

Index

Special characters

- J-Djavac.dump.stack=1 195
- verbose:dynload 347
- verbose:gc (garbage collection) 330
- Xcheckjni 445
- Xtgc:backtrace
 - garbage collection 340
- Xtgc:compaction
 - garbage collection 341
- Xtgc:concurrent
 - garbage collection 341
- Xtgc:dump
 - garbage collection 341
- Xtgc:excessiveGC
 - garbage collection 342
- Xtgc:freelist
 - garbage collection 342
- Xtgc:parallel
 - garbage collection 343
- Xtgc:references
 - garbage collection 343
- Xtgc:scavenger
 - garbage collection 343
- Xtgc:terse
 - garbage collection 344
- Xtrace 195
- /3GB switch, Windows 140
- .dat files 305
- *.nix platforms
 - font utilities 208

Numerics

- 32- and 64-bit JVMs
 - AIX 105
- 32-bit AIX Virtual Memory Model,
 - AIX 105
- 64-bit AIX Virtual Memory Model,
 - AIX 106

A

- ABEND 418
- about this diagnostics guide xiii
- Addr Range, AIX segment type 98
- agent 373
- agent, JVMRI
 - launching 373, 374
 - writing 371
- AIX
 - available disk space 89
 - crashes 100
 - debugging commands 90
 - archon 96
 - band 96
 - cmd 95
 - cp 96
 - dbx Plug-in 99
 - Esid 97
 - f 96

AIX (continued)

- debugging commands (*continued*)
 - netpmn 93
 - netstat 94
 - pid 95
 - ppid 95
 - pri 96
 - ps 94
 - sar 96
 - sc 96
 - st 95
 - stime 95
 - svmon 96
 - tat 96
 - tid 95
 - time 95
 - topas 98
 - trace 98
 - tty 95
 - Type 97
 - uid 95
 - user 95
 - vmstat 99
 - Vsid 97

- debugging hangs 102
 - AIX deadlocks 102
 - busy hangs 102
 - poor performance 105
- debugging memory leaks
 - 32- and 64-bit JVMs 105
 - 32-bit AIX Virtual Memory Model 105
 - 64-bit AIX Virtual Memory Model 106
 - changing the Memory Model (32-bit JVM) 107
 - fragmentation problems 112
 - Java heap exhaustion 111
 - Java or native heap exhausted 111
 - Java2 32-Bit JVM default memory models 108
 - monitoring the Java heap 110
 - monitoring the native heap 108
 - native and Java heaps 108
 - native heap exhaustion 111
 - native heap usage 109
 - receiving OutOfMemory errors 110
 - submitting a bug report 113
- debugging performance
 - problems 113
 - application profiling 119
 - collecting data from a fault condition 119
 - CPU bottlenecks 114
 - finding the bottleneck 113
 - I/O bottlenecks 118
 - JIT compilation 119
 - JVM heap sizing 118
 - memory bottlenecks 118
- debugging techniques 89

AIX (continued)

- diagnosing crashes 100
 - documents to gather 100
 - locating the point of failure 101
- enabling full AIX core files 88
- Java Virtual Machine settings 89
- MALLOCTYPE=watson 110
- operating system settings 88
- problem determination 87
- setting up and checking AIX environment 87
- stack trace 101
 - subpool for garbage collection 16
 - understanding memory usage 105
- allocation failures 333
- analyzing deadlocks, Windows 143
- API calls, JVMRI
 - CreateThread 375
 - DumpDeregister 375
 - DumpRegister 375
 - dynamic verbosegc 376
 - GenerateHeapdump 376
 - GenerateJavacore 376
 - GetComponentDataArea 376
 - GetRasInfo 377
 - InitiateSystemDump 377
 - InjectOutOfMemory 377
 - InjectSigsegv 377
 - NotifySignal 378
 - ReleaseRasInfo 378
 - RunDumpRoutine 378
 - SetOutOfMemoryHook 379
 - TraceDeregister 379
 - TraceDeregister50 379
 - TraceRegister 379
 - TraceRegister50 380
 - TraceResume 380
 - TraceResumeThis 380
 - TraceSet 381
 - TraceSnap 381
 - TraceSuspend 381
 - TraceSuspendThis 381
- application profiling, AIX 119
- application profiling, Linux 133
- application profiling, Windows 146
- application profiling, z/OS 164
- application stack 4
- application trace 306
 - activating and deactivating tracepoints 303
 - example 308
 - printf specifiers 308
 - registering 306
 - suspend or resume 303
 - trace api 303
 - trace buffer snapshot 303
 - tracepoints 307
 - using at runtime 309
- archon, AIX 96

B

- BAD_OPERATION 196
- BAD_PARAM 196
- band, AIX 96
- before you read this book xiv
- bidirectional GIOP, ORB limitation 194
- bottlenecks, AIX
 - CPU 114
 - finding 113
 - I/O 118
 - memory 118
- bottlenecks, Windows
 - finding 146
- bottlenecks, z/OS
 - finding 163
- buffers
 - snapping 286
 - trace 286
- bug report
 - garbage collection 23
- busy hangs, AIX 102

C

- cache allocation (garbage collection) 9
- cache housekeeping
 - shared classes 352
- cache naming
 - shared classes 351
- cache performance
 - shared classes 353
- cache problems
 - shared classes 367, 369
- categorizing problems 217
- CEEDUMPs 229
- changing the Memory Model (32-bit JVM), AIX 107
- checking and setting up environment, Windows 139
- class GC
 - shared classes 354
- class loader 6
 - name spaces and the runtime package 30
 - parent-delegation model 29
 - understanding 29
 - why write your own class loader? 31
- class records in a heapdump 260
- class-loader diagnostics 347
 - command-line options 347
 - loading from native code 348
 - runtime 347
- classic (text) heapdump file format
 - heapdumps 259
- client side interception points, ORB 64
 - receive_exception(receiving reply) 64
 - receive_other(receiving reply) 64
 - receive_reply(receiving reply) 64
 - send_poll(sending request) 64
 - send_request(sending request) 64
- client side, ORB 54
 - getting hold of the remote object 56
 - bootstrap process 57
 - identifying 201
 - ORB initialization 55
 - remote method invocation 57
- client side, ORB (continued)
 - delegation 58
 - stub creation 55
- clnt , AIX segment type 97
- cmd, AIX 95
- codes, minor (CORBA) 409
- collecting data from a fault condition
 - AIX 119
 - Linux 134, 136
 - core files 134
 - determining the operating environment 135
 - proc file system 136
 - producing Javadumps 134
 - producing system dumps 134
 - sending information to Java Support 135
 - strace, ltrace, and mtrace 136
 - using system logs 135
 - Windows 147
 - z/OS 164
- com.ibm.CORBA.AcceptTimeout 51
- com.ibm.CORBA.AllowUserInterrupt 51
- com.ibm.CORBA.BootstrapHost 51
- com.ibm.CORBA.BootstrapPort 51
- com.ibm.CORBA.BufferSize 51
- com.ibm.CORBA.CommTrace 195
- com.ibm.CORBA.ConnectionMultiplicity 51
- com.ibm.CORBA.ConnectTimeout 51
- com.ibm.CORBA.Debug 195
- com.ibm.CORBA.Debug.Output 195
- com.ibm.CORBA.enableLocateRequest 52
- com.ibm.CORBA.FragmentSize 52
- com.ibm.CORBA.FragmentTimeout 52
- com.ibm.CORBA.GIOPAddressingDisposition 52
- com.ibm.CORBA.InitialReferencesURL 52
- com.ibm.CORBA.ListenerPort 52
- com.ibm.CORBA.LocalHost 52
- com.ibm.CORBA.LocateRequestTimeout 52, 203
- com.ibm.CORBA.MaxOpenConnections 53
- com.ibm.CORBA.MinOpenConnections 53
- com.ibm.CORBA.NoLocalInterceptors 53
- com.ibm.CORBA.ORBCharEncoding 53
- com.ibm.CORBA.ORBWCharDefault 53
- com.ibm.CORBA.RequestTimeout 53, 203
- com.ibm.CORBA.SendingContextRunTimeSupportedVersions 53
- com.ibm.CORBA.SendVersionIdentifier 53
- com.ibm.CORBA.ServerSocketQueueDepth 53
- com.ibm.CORBA.ShortExceptionDetails 53
- com.ibm.tools.rmhc.iop.Debug 53
- com.ibm.tools.rmhc.iop.SkipImports 53
- comm trace , ORB 200
- COMM_FAILURE 196
- command-line options 439
 - class-loader 347
 - garbage collector 454
 - general 440
 - JIT 452
 - system property 442
- commands, (IPCS), z/OS 152
- compaction phase (garbage collection)
 - detailed description 15
- compatibility between service releases
 - shared classes 354, 355
- compilation failures, JIT 320
- COMPLETED_MAYBE 196
- COMPLETED_NO 196
- COMPLETED_YES 196
- completion status, ORB 196
- concurrent access
 - shared classes 354
- concurrent mark (garbage collection) 13
- connection handlers 40
- console dumps 228
- control flow optimizations (JIT) 37
- conventions and terminology xv
- CORBA 43
 - client side interception points 64
 - receive_exception(receiving reply) 64
 - receive_other(receiving reply) 64
 - receive_reply(receiving reply) 64
 - send_poll(sending request) 64
 - send_request(sending request) 64
 - examples 45
 - fragmentation 63
 - further reading 45
 - interfaces 45
 - Interoperable Naming Service (INS) 66
 - Java IDL or RMI-IIOP, choosing 44
 - minor codes 409
 - portable interceptors 63
 - portable object adapter 61
 - remote object implementation (or servant) 45
 - RMI and RMI-IIOP 44
 - RMI-IIOP limitations 44
 - server code 47
 - differences between RMI (JRMP) and RMI-IIOP 50
 - summary of differences in client development 50
 - summary of differences in server development 50
 - server side interception points 64
 - receive_request_service_contexts(receiving request) 64
 - receive_request(receiving request) 64
 - send_exception(sending reply) 64
 - send_other(sending reply) 64
 - send_reply(sending reply) 64
 - stubs and ties generation 46
 - core dump 263
 - defaults 264
 - overview 263
- core files
 - Linux 121
- core files, Linux 134
- cp, AIX 96
- CPU bottlenecks, AIX 114
- CPU usage, Linux 131
- crashes
 - AIX 100
 - Linux 129
 - Windows 142
 - z/OS 153
 - documents to gather 153
 - failing function 154

- crashes, diagnosing
 - Windows
 - sending data to IBM 142
- CreateThread, JVMRI 375
- cross-platform tools
 - DTFJ 220
 - dump viewer 219
 - Heapdump 219
 - JPDA tools 220
 - JVMPI tools 220
 - JVMRI 221
 - JVMTI 219
 - trace formatting 220

D

- DATA_CONVERSION 196
- dbx Plug-in, AIX 99
- deadlocked process, z/OS 160
- deadlocks 102, 252
- deadlocks, Windows
 - debugging 143
- debug properties, ORB 194
 - com.ibm.CORBA.CommTrace 195
 - com.ibm.CORBA.Debug 195
 - com.ibm.CORBA.Debug.Output 195
- debugging commands
 - AIX 90
 - dbx Plug-in 99
 - netpmon 93
 - netstat 94
 - sar 96
 - topas 98
 - trace 98
 - vmstat 99
- debugging hangs, AIX 102
 - AIX deadlocks 102
 - busy hangs 102
 - poor performance 105
- debugging hangs, Windows 143
- debugging memory leaks, AIX
 - 32- and 64-bit JVMs 105
 - 32-bit AIX Virtual Memory Model 105
 - 64-bit AIX Virtual Memory Model 106
 - changing the Memory Model (32-bit JVM) 107
 - fragmentation problems 112
 - Java heap exhaustion 111
 - Java or native heap exhausted 111
 - Java2 32-Bit JVM default memory models 108
 - monitoring the Java heap 110
 - monitoring the native heap 108
 - native and Java heaps 108
 - native heap exhaustion 111
 - native heap usage 109
 - receiving OutOfMemory errors 110
 - submitting a bug report 113
- debugging memory leaks, Windows
 - memory model 144
 - tracing leaks 144
- debugging performance problem, AIX
 - application profiling 119
 - collecting data from a fault condition 119
- debugging performance problem, AIX (*continued*)
 - CPU bottlenecks 114
 - finding the bottleneck 113
 - I/O bottlenecks 118
 - JIT compilation 119
 - JVM heap sizing 118
 - memory bottlenecks 118
- debugging performance problem, Linux
 - JIT compilation 133
 - JVM heap sizing 133
- debugging performance problem, Linux
 - application profiling 133
- debugging performance problem, Windows
 - application profiling 146
 - finding the bottleneck 146
 - JIT compilation 146
 - JVM heap sizing 146
 - systems resource usage 146
- debugging performance problem, z/OS
 - application profiling 164
 - finding the bottleneck 163
 - JIT compilation 164
 - JVM heap sizing 164
 - systems resource usage 164
- debugging performance problems, AIX 113
- debugging performance problems, Linux
 - CPU usage 131
 - finding the bottleneck 131
 - memory usage 132
 - network problems 132
- debugging performance problems, Windows 145
- debugging techniques, AIX 89
 - dbx Plug-in 99
 - debugging commands 90
 - netpmon 93
 - netstat 94
 - sar 96
 - topas 98
 - trace 98
 - vmstat 99
- debugging techniques, Linux
 - ldd command 126
 - ps command 124
- default memory models, Java2 32-Bit JVM (AIX) 108
- default settings, JVM 461
- defaults
 - core dump 264
- delegation, ORB client side 58
- deploying shared classes 351
- deprecated Sun properties 54
- description string, ORB 199
- Description, AIX segment type 97
- determining the operating environment, Linux 135
- df command, Linux 135
- diagnosing crashes, AIX 100
 - documents to gather 100
 - locating the point of failure 101
- Diagnostics Collector 323
- diagnostics component 5
- diagnostics options, JVM
 - environment 414
- diagnostics, class loader
 - loading from native code 348
 - runtime 347
- diagnostics, class-loader 347
 - command-line options 347
- diagnostics, overview 217
 - categorizing problems 217
 - cross-platform tools 219
 - DTFJ 220
 - dump viewer 219
 - Heapdump 219
 - JPDA tools 220
 - JVMPI tools 220
 - JVMRI 221
 - JVMTI 219
 - trace formatting 220
- differences between RMI (JRMP) and RMI-IIOP, ORB 50
- disabling the JIT compiler 317
- Distributed Garbage Collection (DGC)
 - RMI 41
- documents to gather
 - AIX 100
- DTFJ
 - counting threads example 397
 - diagnostics 393
 - example of the interface 393
 - interface diagram 396
 - working with a dump 394
- DTFJ, cross-platform tools 220
- dump
 - core 263
 - defaults 264
 - overview 263
 - signals 241
 - z/OS 242
- dump agents
 - CEEDUMPs 229
 - console dumps 228
 - default 238
 - environment variables 240
 - events 233
 - filters 234
 - heapdumps 231
 - Java dumps 231
 - removing 239
 - snap traces 232
 - stack dumps 229
 - system dumps 228
 - tool option 230
 - using 223
- dump viewer 263
 - analyzing dumps 273
 - cross-platform tools 219
 - example session 273
- DumpDeregister, JVMRI 375
- DumpRegister, JVMRI 375
- dumps, setting up (z/OS) 150
- dynamic updates
 - shared classes 358
- dynamic verbosegc, JVMRI 376

E

- enabling full AIX core files 88
- environment
 - displaying current 413

- environment (*continued*)
 - JVM settings 414
 - deprecated JIT options 414
 - diagnostics options 414
 - general options 414
 - Javadump and Heapdump options 414
 - setting up and checking on Windows 139
- environment variables 413
 - dump agents 240
 - heapdumps 258
 - javadumps 256
 - separating values in a list 414
 - setting 413
 - z/OS 149, 418
- environment, determining
 - Linux 135
 - df command 135
 - free command 135
 - lsdf command 135
 - ps-ef command 135
 - top command 135
 - uname -a command 135
 - vmstat command 135
- error message IDs
 - z/OS 153
- errors (OutOfMemory), receiving (AIX) 110
- Esid, AIX 97
- events
 - dump agents 233
- example of real method trace 314
- examples of method trace 313
- exceptions, JNI 76
- exceptions, ORB 195
 - completion status and minor codes 196
- system 196
 - BAD_OPERATION 196
 - BAD_PARAM 196
 - COMM_FAILURE 196
 - DATA_CONVERSION 196
 - MARSHAL 196
 - NO_IMPLEMENT 196
 - UNKNOWN 196
- user 195
- exhaustion of Java heap, AIX 111
- exhaustion of native heap, AIX 111

F

- f, AIX 96
- failing function, z/OS 154
- failing method, JIT 319
- fault condition in AIX
 - collecting data from 119
- file header, Javadump 248
- finalizers 330
- finding classes
 - shared classes 359
- finding the bottleneck, Linux 131
- first steps in problem determination 85
- floating stacks limitations, Linux 136
- font limitations, Linux 137
- fonts, NLS 207
 - common problems 209

- fonts, NLS (*continued*)
 - installed 208
 - properties 207
 - utilities
 - *.nix platforms 208
- formatting, JVMRI 383
- fragmentation
 - AIX 112
 - ORB 63, 194
- free command, Linux 135
- frequently asked questions
 - garbage collection 26
 - JIT 37
- functions (table), JVMRI 374

G

- garbage collection 8
 - advanced diagnostics
 - Xtgc:backtrace 340
 - Xtgc:compaction 341
 - Xtgc:concurrent 341
 - Xtgc:dump 341
 - Xtgc:excessiveGC 342
 - Xtgc:freelist 342
 - Xtgc:parallel 343
 - Xtgc:references 343
 - Xtgc:scavenger 343
 - Xtgc:terse 344
 - TGC tracing 340
 - allocation failures 333
 - allocation failures during concurrent mark 336
 - basic diagnostics (verbose:gc) 330
 - cache allocation 9
 - coexisting with the Garbage Collector 23
 - bug reports 23
 - finalizers 24
 - finalizers and the garbage collection contract 24
 - finalizers, summary 25
 - how finalizers are run 25
 - manual invocation 25
 - nature of finalizers 24
 - thread local heap 23
 - command-line options 454
 - common causes of perceived leaks 329
 - hash tables 330
 - JNI references 330
 - listeners 329
 - objects with finalizers 330
 - static data 330
 - compaction phase
 - detailed description 15
 - concurrent 336
 - concurrent kickoff 336
 - concurrent mark 13
 - concurrent sweep completed 336
 - detailed description 11
 - fine tuning options 22
 - frequently asked questions 26
 - Generational Concurrent Garbage Collector 19
 - global collections 331
 - heap expansion 17

- garbage collection (*continued*)
 - heap lock allocation 9
 - heap shrinkage 18
 - heap size
 - problems 8
 - how does it work? 329
 - how to do heap sizing 21
 - initial and maximum heap sizes 21
 - interaction with applications 22
 - interaction with JNI 70
 - JNI weak reference 17
 - Large Object Area 10
 - mark phase
 - detailed description 11
 - mark stack overflow 12
 - parallel mark 12
 - memory allocation 9
 - nursery allocation failures 333
 - object allocation 7
 - output from a System.gc() 332
 - overview 7
 - parallel bitwise sweep 14
 - phantom reference 16
 - reachable objects 8
 - reference objects 16
 - scavenger collections 334
 - soft reference 16
 - subpool 16
 - sweep phase
 - detailed description 14
 - System.gc() calls during concurrent mark 339
 - tenure age 20
 - tenured allocation failures 334
 - tilt ratio 20
 - timing problems 339
 - understanding the Garbage Collectors 7
 - using verbose:gc 21
 - verbose, heap information 258
 - weak reference 16
- gdb 127
- GenerateHeapdump, JVMRI 376
- GenerateJavacore, JVMRI 376
- Generational Concurrent Garbage Collector
 - sizing, garbage collection 19
- Generational Garbage Collector
 - tenure age 20
 - tilt ratio 20
- GetComponentDataArea, JVMRI 376
- GetRasInfo, JVMRI 377
- getting a dump from a hung JVM, Windows 143
- glibc limitations, Linux 137
- global optimizations (JIT) 37
- growing classpaths
 - shared classes 354

H

- hanging, ORB 202
 - com.ibm.CORBA.LocateRequestTimeout 203
 - com.ibm.CORBA.RequestTimeout 203
- hangs
 - AIX
 - busy hangs 102

- hangs (*continued*)
 - Windows
 - debugging 143
 - z/OS 159
 - bad performance 161
- hangs, debugging
 - AIX 102
 - AIX deadlocks 102
 - poor performance 105
- hash tables 330
- header record in a heapdump 259
- heap
 - expansion 17
 - lock allocation 9
 - shrinkage 18
 - size, garbage collection
 - problems 8
 - sizing, garbage collection 21
- heap (Java) exhaustion, AIX 111
- heap, verbose GC 258
- heapdump
 - Linux 123
- Heapdump 257
 - cross-platform tools 219
 - enabling 257
 - environment variables 258
 - previous releases 257
 - text (classic) Heapdump file
 - format 259
- heapdumps 231
- heaps, native and Java
 - AIX 108
- Hewlett-Packard
 - problem determination 191
- how to read this book xiv
- HPROF Profiler 385
 - options 385
 - output file 386
- hung JVM
 - getting a dump from
 - Windows 143

I

- I/O bottlenecks, AIX 118
- IBM_JAVA_ABEND_ON_FAILURE 418
- initialization problems
 - shared classes 367
- InitiateSystemDump, JVMRI 377
- InjectOutOfMemory, JVMRI 377
- InjectSigsegv, JVMRI 377
- inlining (JIT) 36
- INS, ORB 66
- interceptors (portable), ORB 63
- Interface Definition Language (IDL) 44
- Interoperable Naming Service (INS), ORB 66
- interpreter 6
- interpreting the stack trace, AIX 101
- Inuse, AIX segment type 98
- IPCS commands, z/OS 152

J

- Java archive and compressed files
 - shared classes 353

- Java dumps 231
- Java heap, AIX 108
 - exhaustion 111
 - monitoring 110
- Java Helper API
 - shared classes 361
- Java Native Interface
 - see JNI 69
- Java or native heap exhausted, AIX 111
- JAVA_DUMP_OPTS 418
 - default dump agents 238
 - JVMRI 377
 - parsing 240
 - setting up dumps 150
- JAVA_LOCAL_TIME 418
- JAVA_TDUMP_PATTERN=string 418
- JAVA_THREAD_MODEL 418
- Java2 32-Bit JVM default memory models, AIX 108
- Javadump 245
 - enabling 245
 - environment variables 256
 - file header, gpinfo 248
 - file header, title 248
 - interpreting 247
 - Linux 123
 - Linux, producing 134
 - locks, monitors, and deadlocks (LOCKS) 252
 - storage management 250
 - system properties 248
 - tags 247
 - threads and stack trace (THREADS) 253, 254
 - triggering 245
- jdumpview 263, 266
 - commands 267
 - dump details 269
 - general 267
 - heapdump 271
 - locks 269
 - memory analysis 269
 - trace 272
 - working with classes 270
 - working with objects 271
 - example session 273
- jextract 264
 - overview 264
- jextract 264
- JIT
 - command-line options 452
 - compilation failures, identifying 320
 - control flow optimizations 37
 - disabling 317
 - frequently asked questions 37
 - global optimizations 37
 - how the JIT optimizes code 36
 - idle 322
 - inlining 36
 - JVM environment options 414
 - local optimizations 36
 - locating the failing method 319
 - native code generation 37
 - ORB-connected problem 194
 - overview 35
 - problem determination 317
 - selectively disabling 318

- JIT (*continued*)
 - short-running applications 321
 - understanding 35
- JIT compilation
 - AIX 119
 - Linux 133
 - Windows 146
 - z/OS 164
- JNI 69
 - checklist 79
 - copying and pinning 75
 - debugging 78
 - exceptions 76
 - garbage collection 17
 - generic use of isCopy and mode
 - flags 76
 - interaction with Garbage
 - Collector 70
 - isCopy flag 75
 - mode flag 76
 - problem determination 78
 - references for garbage collection 330
 - synchronization 77
 - understanding 69
 - weak reference 17
- JPDA tools, cross-platform tools 220
- JVM
 - API 5
 - application stack 4
 - building blocks 3
 - class loader 6
 - components 4
 - diagnostics component 5
 - environment settings 414
 - deprecated JIT options 414
 - diagnostics options 414
 - general options 414
 - Javadump and Heapdump
 - options 414
 - interpreter 6
 - memory management 5
 - platform port layer 6
 - trace formatting 220
 - JVM dump initiation
 - locations 241
 - z/OS 242
 - JVM heap sizing
 - AIX 118
 - Linux 133
 - Windows 146
 - z/OS 164
 - JVMPI
 - cross-platform tools 220
 - JVMRI 371
 - agent design 374
 - API calls 375
 - CreateThread 375
 - DumpDeregister 375
 - DumpRegister 375
 - dynamic verbosegc 376
 - GenerateHeapdump 376
 - GenerateJavacore 376
 - GetComponentDataArea 376
 - GetRasInfo 377
 - InitiateSystemDump 377
 - InjectOutOfMemory 377
 - InjectSigsegv 377

JVMRI (continued)

API calls (continued)

- NotifySignal 378
- ReleaseRasInfo 378
- RunDumpRoutine 378
- SetOutOfMemoryHook 379
- TraceDeregister 379
- TraceDeregister50 379
- TraceRegister 379
- TraceRegister50 380
- TraceResume 380
- TraceResumeThis 380
- TraceSet 381
- TraceSnap 381
- TraceSuspend 381
- TraceSuspendThis 381
- changing trace options 373
- cross-platform tools 221
- formatting 383
- functions (table) 374
- launching the agent 373, 374
- RasInfo
 - request types 382
 - structure 382
- registering a trace listener 372
- writing an agent 371

JVMTI

- cross-platform tools 219
- diagnostics 391

K

- kernel, AIX segment type 97
- known limitations, Linux 136
 - floating stacks limitations 136
 - font limitations 137
 - glibc limitations 137
 - threads as processes 136

L

- large address aware support, Windows 140
- Large Object Area (garbage collection) 10
- ldd command 126
- LE HEAP, z/OS 161
- LE settings, z/OS 149
- limitations, Linux 136
 - floating stacks limitations 136
 - font limitations 137
 - glibc limitations 137
 - threads as processes 136
- Linux
 - collecting data from a fault condition 134, 136
 - core files 134
 - determining the operating environment 135
 - proc file system 136
 - producing Javadumps 134
 - producing system dumps 134
 - sending information to Java Support 135
 - strace, ltrace, and mtrace 136
 - using system logs 135

Linux (continued)

- core files 121
 - crashes, diagnosing 129
 - debugging commands
 - gdb 127
 - ltrace tool 126
 - mtrace tool 126
 - strace tool 126
 - tracing tools 126
 - debugging hangs 130
 - debugging memory leaks 131
 - debugging performance problems 131
 - application profiling 133
 - CPU usage 131
 - finding the bottleneck 131
 - JIT compilation 133
 - JVM heap sizing 133
 - memory usage 132
 - network problems 132
 - debugging techniques 123
 - known limitations 136
 - floating stacks limitations 136
 - font limitations 137
 - glibc limitations 137
 - threads as processes 136
 - ldd command 126
 - ltrace 136
 - mtrace 136
 - nm command 124
 - objdump command 124
 - problem determination 121
 - ps command 124
 - setting up and checking the environment 121
 - starting heapdumps 123
 - starting Javadumps 123
 - strace 136
 - threading libraries 123
 - top command 125
 - tracing tools 126
 - using system dumps 124
 - using system logs 124
 - vmstat command 125
 - working directory 121
- listeners 329
- local optimizations (JIT) 36
- locating the failing method, JIT 319
- locks, monitors, and deadlocks (LOCKS), Javaldump 252
- looping process, z/OS 160
- lsof command, Linux 135
- ltrace, Linux 136

M

- maintenance, z/OS 149
- MALLOCTYPE=watson 110
- mark phase (garbage collection)
 - concurrent mark 13
 - detailed description 11
 - parallel mark 12
- MARSHAL 196
- memory allocation 9
 - cache allocation 9
 - Large Object Area 10
- memory bottlenecks, AIX 118

memory leaks

Windows

- classifying 144
- debugging 143
- z/OS 161
 - LE HEAP 161
 - OutOfMemoryErrors 162
 - virtual storage 161

memory leaks, debugging

AIX

- 32- and 64-bit JVMs 105
- 32-bit AIX Virtual Memory Model 105
- 64-bit AIX Virtual Memory Model 106
- changing the Memory Model (32-bit JVM) 107
- Java heap exhaustion 111
- Java or native heap exhausted 111
- Java2 32-Bit JVM default memory models 108
- monitoring the Java heap 110
- monitoring the native heap 108
- native and Java heaps 108
- native heap exhaustion 111
- native heap usage 109
- receiving OutOfMemory errors 110

memory leaks, Windows

tracing 144

memory management 5

- heap lock allocation 9
- how to do heap sizing 21
- memory allocation 9

Memory Model (32-bit JVM), changing, AIX 107

memory model, Windows 144

memory models, Java2 32-Bit JVM default (AIX) 108

memory usage, Linux 132

memory usage, understanding AIX 105

message trace , ORB 199

method trace 309

- examples 313
- real example 314
- running with 309

minor codes , CORBA 409

minor codes, ORB 196

mmap, AIX segment type 97

modification contexts

shared classes 357

monitoring the Java heap, AIX 110

monitoring the native heap, AIX 108

monitors, Javaldump 252

mtrace, Linux 136

N

native code generation (JIT) 37

native heap, AIX 108

- exhaustion 111
- monitoring 108
- usage 109

netpmmon, AIX 93

netstat, AIX 94

network problems, Linux 132

- NLS
 - font properties 207
 - fonts 207
 - installed fonts 208
 - problem determination 207
- NO_IMPLEMENT 196
- non-standard 444
- nonstandard 444
- NotifySignal, JVMRI 378

O

- object allocation 7
- object records in a heapdump 259
- objects
 - reachable 8
- objects with finalizers 330
- options
 - command-line 439
 - general 440
 - system property 442
 - JVM environment
 - deprecated JIT 414
 - diagnostics 414
 - general 414
- ORB 43
 - additional features 61
 - bidirectional GIOP limitation 194
 - choosing Java IDL or RMI-IIOP 44
 - client side 54
 - bootstrap process 57
 - delegation 58
 - getting hold of the remote object 56
 - ORB initialization 55
 - remote method invocation 57
 - stub creation 55
 - common problems 202
 - client and server running, not naming service 204
 - com.ibm.CORBA.LocateRequestTimeout 203
 - com.ibm.CORBA.RequestTimeout 203
 - hanging 202
 - running the client with client unplugged 204
 - running the client without server 203
 - completion status and minor codes 196
 - component, what it contains 193
- CORBA
 - differences between RMI (JRMP) and RMI-IIOP 50
 - examples 45
 - further reading 45
 - interfaces 45
 - introduction 43
 - Java IDL or RMI-IIOP? 44
 - remote object implementation (or servant) 45
 - RMI-IIOP limitations 44
 - server code 47
 - stubs and ties generation 46
 - summary of differences in client development 50
 - summary of differences in server development 50

- ORB (*continued*)
 - debug properties 194
 - com.ibm.CORBA.CommTrace 195
 - com.ibm.CORBA.Debug 195
 - com.ibm.CORBA.Debug.Output 195
 - debugging 193
 - diagnostic tools
 - J-Djavac.dump.stack=1 195
 - Xtrace 195
 - exceptions 195
 - features
 - client side interception points 64
 - fragmentation 63
 - Interoperable Naming Service (INS) 66
 - portable interceptors 63
 - portable object adapter 61
 - server side interception points 64
 - how it works 54
 - identifying a problem 193
 - fragmentation 194
 - JIT problem 194
 - ORB versions 194
 - platform-dependent problem 194
 - what the ORB component contains 193
 - properties 51
- RMI and RMI-IIOP
 - differences between RMI (JRMP) and RMI-IIOP 50
 - examples 45
 - further reading 45
 - interfaces 45
 - introduction 44
 - remote object implementation (or servant) 45
 - server code 47
 - stub and ties generation 46
 - summary of differences in client development 50
 - summary of differences in server development 50
- RMI-IIOP limitations 44
- security permissions 197
- server side 59
 - processing a request 60
 - servant binding 59
 - servant implementation 59
 - tie generation 59
- service: collecting data 205
- preliminary tests 205
- stack trace 198
 - description string 199
- system exceptions 196
 - BAD_OPERATION 196
 - BAD_PARAM 196
 - COMM_FAILURE 196
 - DATA_CONVERSION 196
 - MARSHAL 196
 - NO_IMPLEMENT 196
 - UNKNOWN 196
- traces 199
 - client or server 201
 - comm 200
 - message 199
 - service contexts 202

- ORB (*continued*)
 - understanding
 - additional features 61
 - client side interception points 64
 - fragmentation 63
 - how it works 54
 - Interoperable Naming Service (INS) 66
 - portable interceptors 63
 - portable object adapter 61
 - processing a request 60
 - servant binding 59
 - servant implementation 59
 - server side interception points 64
 - the client side 54
 - the server side 59
 - tie generation 59
 - using 51
 - user exceptions 195
 - versions 194
- ORB properties
 - com.ibm.CORBA.AcceptTimeout 51
 - com.ibm.CORBA.AllowUserInterrupt 51
 - com.ibm.CORBA.BootstrapHost 51
 - com.ibm.CORBA.BootstrapPort 51
 - com.ibm.CORBA.BufferSize 51
 - com.ibm.CORBA.ConnectionMultiplicity 51
 - com.ibm.CORBA.ConnectTimeout 51
 - com.ibm.CORBA.enableLocateRequest 52
 - com.ibm.CORBA.FragmentSize 52
 - com.ibm.CORBA.FragmentTimeout 52
 - com.ibm.CORBA.GIOPAddressingDisposition 52
 - com.ibm.CORBA.InitialReferencesURL 52
 - com.ibm.CORBA.ListenerPort 52
 - com.ibm.CORBA.LocalHost 52
 - com.ibm.CORBA.LocateRequestTimeout 52
 - com.ibm.CORBA.MaxOpenConnections 53
 - com.ibm.CORBA.MinOpenConnections 53
 - com.ibm.CORBA.NoLocalInterceptors 53
 - com.ibm.CORBA.ORBCharEncoding 53
 - com.ibm.CORBA.ORBWCharDefault 53
 - com.ibm.CORBA.RequestTimeout 53
 - com.ibm.CORBA.SendingContextRunTimeSupported 53
 - com.ibm.CORBA.SendVersionIdentifier 53
 - com.ibm.CORBA.ServerSocketQueueDepth 53
 - com.ibm.CORBA.ShortExceptionDetails 53
 - com.ibm.tools.rmic.iioop.Debug 53
 - com.ibm.tools.rmic.iioop.SkipImports 53
 - org.omg.CORBA.ORBId 53
 - org.omg.CORBA.ORBListenEndpoints 54
 - org.omg.CORBA.ORBServerId 54
 - org.omg.CORBA.ORBId 53
 - org.omg.CORBA.ORBListenEndpoints 54
 - org.omg.CORBA.ORBServerId 54
- OSGi ClassLoading Framework
 - shared classes 371
- other sources of information xv
- OutOfMemory errors, receiving (AIX) 110
- OutOfMemoryErrors, z/OS 162
- overview of diagnostics 217
 - categorizing problems 217
 - cross-platform tools 219
 - DTFJ 220
 - dump viewer 219
 - Heapdump 219
 - JPDA tools 220

- overview of diagnostics *(continued)*
 - cross-platform tools *(continued)*
 - JVMPI tools 220
 - JVMRI 221
 - JVMTI 219
 - trace formatting 220

P

- parallel mark (garbage collection) 12
- parent-delegation model (class loader) 29
- performance problems, debugging
 - AIX 113
 - application profiling 119
 - collecting data from a fault condition 119
 - CPU bottlenecks 114
 - finding the bottleneck 113
 - I/O bottlenecks 118
 - JIT compilation 119
 - JVM heap sizing 118
 - memory bottlenecks 118
 - Linux
 - application profiling 133
 - CPU usage 131
 - finding the bottleneck 131
 - JIT compilation 133
 - JVM heap sizing 133
 - memory usage 132
 - network problems 132
 - Windows 145
 - application profiling 146
 - finding the bottleneck 146
 - JIT compilation 146
 - JVM heap sizing 146
 - systems resource usage 146
 - z/OS 163
 - application profiling 164
 - badly-performing process 161
 - finding the bottleneck 163
 - JIT compilation 164
 - JVM heap sizing 164
 - systems resource usage 164
- pers, AIX segment type 97
- Pgsp, AIX segment type 98
- pid, AIX 95
- Pin, AIX segment type 98
- platform-dependent problem, ORB 194
- poor performance, AIX 105
- portable interceptors, ORB 63
- portable object adapter
 - ORB 61
- power management 286
- ppid, AIX 95
- preliminary tests for collecting data, ORB 205
- pri, AIX 96
- printAllStats utility
 - shared classes 365
- printStats utility
 - shared classes 364
- private storage usage, z/OS 149
- problem determination
 - Hewlett-Packard 191
 - Sun Solaris 189
- problems, ORB 202

- problems, ORB *(continued)*
 - hanging 202
- proc file system, Linux 136
- process
 - z/OS
 - deadlocked 160
 - looping 160
- process private, AIX segment type 98
- producing Javadumps, Linux 134
- producing system dumps, Linux 134
- properties, ORB 51
 - com.ibm.CORBA.AcceptTimeout 51
 - com.ibm.CORBA.AllowUserInterrupt 51
 - com.ibm.CORBA.BootstrapHost 51
 - com.ibm.CORBA.BootstrapPort 51
 - com.ibm.CORBA.BufferSize 51
 - com.ibm.CORBA.ConnectionMultiplicity 51
 - com.ibm.CORBA.ConnectTimeout 51
 - com.ibm.CORBA.enableLocateRequest 52
 - com.ibm.CORBA.FragmentSize 52
 - com.ibm.CORBA.FragmentTimeout 52
 - com.ibm.CORBA.GIOPAddressingDisposition 52
 - com.ibm.CORBA.InitialReferencesURL 52
 - com.ibm.CORBA.ListenerPort 52
 - com.ibm.CORBA.LocalHost 52
 - com.ibm.CORBA.LocateRequestTimeout 52
 - com.ibm.CORBA.MaxOpenConnections 53
 - com.ibm.CORBA.MinOpenConnections 53
 - com.ibm.CORBA.NoLocalInterceptors 53
 - com.ibm.CORBA.ORBCharEncoding 53
 - com.ibm.CORBA.ORBWCharDefault 53
 - com.ibm.CORBA.RequestTimeout 53
 - com.ibm.CORBA.SendingContextRunTimeSupported 53
 - com.ibm.CORBA.SendVersionIdentifier 53
 - com.ibm.CORBA.ServerSocketQueueDepth 53
 - com.ibm.CORBA.ShortExceptionDetails 53
 - com.ibm.tools.rmhc.iioop.Debug 53
 - com.ibm.tools.rmhc.iioop.SkipImports 53
 - org.omg.CORBA.ORBId 53
 - org.omg.CORBA.ORBListenEndpoints 54
 - org.omg.CORBA.ORBServerId 54
- ps command
 - AIX 94
- ps-ef command, Linux 135

R

- RAS interface (JVMRI) 371
- RasInfo, JVMRI
 - request types 382
 - structure 382
- receive_exception(receiving reply) 64
- receive_other(receiving reply) 64
- receive_reply(receiving reply) 64
- receive_request_service_contexts(receiving request) 64
- receive_request(receiving request) 64
- receiving OutOfMemory errors, AIX 110
- redeeming stale classes
 - shared classes 360
- reference objects (garbage collection) 16
- ReleaseRasInfo, JVMRI 378
- reliability, availability, and serviceability interface (JVMRI) 371
- Remote Method Invocation
 - See RMI 39

- remote object
 - ORB client side
 - bootstrap process 57
 - getting hold of 56
 - remote method invocation 57
- remote object implementation (or servant)
 - ORB 45
- ReportEnv
 - AIX 88
 - Linux 121
 - Windows 139
- reporting problems in the JVM, summary xv
- request types, JVMRI (RasInfo) 382
- RMI 39
 - debugging applications 41
 - Distributed Garbage Collection (DGC) 41
 - examples 45
 - further reading 45
 - implementation 39
 - interfaces 45
 - introduction 44
 - remote object implementation (or servant) 45
 - server code 47
 - differences between RMI (JRMP) and RMI-IIOP 50
 - summary of differences in client development 50
 - summary of differences in server development 50
 - stubs and ties generation 46
 - thread pooling 40
 - RMI-IIOP
 - choosing against Java IDL 44
 - examples 45
 - further reading 45
 - interfaces 45
 - introduction 44
 - limitations 44
 - remote object implementation (or servant) 45
 - server code 47
 - differences between RMI (JRMP) and RMI-IIOP 50
 - summary of differences in client development 50
 - summary of differences in server development 50
 - stubs and ties generation 46
- RunDumpRoutine, JVMRI 378
- runtime bytecode modification
 - shared classes 356
- runtime diagnostics, class loader 347

S

- Safemode
 - shared classes 357
- sar, AIX 96
- sc, AIX 96
- security permissions for the ORB 197
- see also jdmpview 263
- selectively disabling the JIT 318
- send_exception(sending reply) 64
- send_other(sending reply) 64

- send_poll(sending request) 64
- send_reply(sending reply) 64
- send_request(sending request) 64
- sending data to IBM, Windows 142
- sending information to Java Support, Linux 135
- server code, ORB 47
- server side interception points, ORB 64
 - receive_request_service_contexts(receiving request) 64
 - receive_request(receiving request) 64
 - send_exception(sending reply) 64
 - send_other(sending reply) 64
 - send_reply(sending reply) 64
- server side, ORB 59
 - identifying 201
 - processing a request 60
 - servant binding 59
 - servant implementation 59
 - tie generation 59
- service contexts, ORB 202
- service: collecting data, ORB 205
 - preliminary tests 205
- SetOutOfMemoryHook, JVMRI 379
- setting up and checking AIX environment 87
- setting up and checking environment, Windows 139
- settings, default (JVM) 461
- settings, JVM
 - environment 414
 - deprecated JIT options 414
 - diagnostics options 414
 - general options 414
 - Javdump and Heapdump options 414
- shared classes
 - benefits 33
 - cache housekeeping 352
 - cache naming 351
 - cache performance 353
 - cache problems 367, 369
 - class GC 354
 - compatibility between service releases 354, 355
 - concurrent access 354
 - deploying 351
 - diagnostics 351
 - diagnostics output 363
 - dynamic updates 358
 - finding classes 359
 - growing classpaths 354
 - initialization problems 367
 - introduction 33
 - Java archive and compressed files 353
 - Java Helper API 361
 - modification contexts 357
 - not filling the cache 353
 - OSGi ClassLoading Framework 371
 - printAllStats utility 365
 - printStats utility 364
 - problem debugging 366
 - redeeming stale classes 360
 - runtime bytecode modification 356
 - Safemode 357
 - SharedClassHelper partitions 357

- shared classes (*continued*)
 - stale classes 360
 - storing classes 359
 - trace 366
 - verbose output 363
 - verboseHelper output 364
 - verboseIO output 363
 - verification problems 369
- shared library, AIX segment type 98
- SharedClassHelper partitions
 - shared classes 357
- shmat/mmap, AIX segment type 98
- short-running applications
 - JIT 321
- skeletons, ORB 46
- snap traces 232
- st, AIX 95
- stack dumps 229
- stack trace, interpreting (AIX) 101
- stack trace, ORB 198
 - description string 199
- stale classes
 - shared classes 360
- start 373
- starting 373
- static data 330
- stime, AIX 95
- storage management, Javdump 250
- storage usage, private (z/OS) 149
- storage, z/OS 161
- storing classes
 - shared classes 359
- strace, Linux 136
- string (description), ORB 199
- stusb and ties generation, ORB 46
- submitting a bug report, AIX 113
- summary of differences in client development 50
- summary of differences in server development 50
- Sun properties, deprecated 54
- Sun Solaris
 - problem determination 189
- svmon, AIX 96
- sweep phase (garbage collection)
 - detailed description 14
 - parallel bitwise sweep 14
- synchronization
 - JNI 77
- system dump 263
 - defaults 264
 - overview 263
 - Windows 141
- System dump
 - Linux, producing 134
- system dumps 228
 - Linux 124
- system exceptions, ORB 196
 - BAD_OPERATION 196
 - BAD_PARAM 196
 - COMM_FAILURE 196
 - DATA_CONVERSION 196
 - MARSHAL 196
 - NO_IMPLEMENT 196
 - UNKNOWN 196
- system logs 124
- system logs, using (Linux) 135

- system properties
 - command-line options 442
- system properties, Javdump 248
- System.gc() 332, 339
- systems resource usage, Windows 146
- systems resource usage, z/OS 164

T

- tags, Javdump 247
- tat, AIX 96
- TDUMPs
 - z/OS 156
- tenure age 20
- terminology and conventions xv
- text (classic) heapdump file format
 - heapdumps 259
- TGC tracing
 - garbage collection 340
- thread pooling
 - RMI 40
- threading libraries 123
- threads and stack trace (THREADS) 253, 254
- threads as processes, Linux 136
- tid, AIX 95
- tilt ratio 20
- time, AIX 95
- timing problems in garbage collection 339
- tool option for dumps 230
- tools
 - cross-platform 219
- tools, ReportEnv
 - AIX 88
 - Linux 121
 - Windows 139
- top command, Linux 135
- topas, AIX 98
- trace
 - .dat files 305
 - AIX 98
 - application trace 306
 - applications 283
 - changing options 373
 - controlling 287
 - default 284
 - default assertion tracing 285
 - default memory management tracing 285
 - formatter 304
 - invoking 304
 - intercepting trace data 382
 - internal 283
 - Java applications and the JVM 283
 - methods 283
 - options
 - buffers 290
 - count 291
 - detailed descriptions 288
 - exception 291
 - exception.output 299
 - external 291
 - iprint 291
 - maximal 291
 - method 296
 - minimal 291

- trace (*continued*)
 - options (*continued*)
 - none 289
 - output 298
 - print 291
 - properties 289
 - resume 300
 - resumecount 300
 - specifying 288
 - suspend 300
 - suspendcount 301
 - trigger 301
 - placing data into a file 286
 - external tracing 287
 - trace combinations 287
 - tracing to stderr 287
 - placing data into memory
 - buffers 286
 - snapping buffers 286
 - power management effect on
 - timers 286
 - registering a trace listener 372
 - shared classes 366
 - tracepoint ID 305
- TraceDeregister, JVMRI 379
- TraceDeregister50, JVMRI 379
- tracepoint specification 293
- TraceRegister, JVMRI 379
- TraceRegister50, JVMRI 380
- TraceResume, JVMRI 380
- TraceResumeThis, JVMRI 380
- traces, ORB 199
 - client or server 201
 - comm 200
 - message 199
 - service contexts 202
- TraceSet, JVMRI 381
- TraceSnap, JVMRI 381
- TraceSuspend, JVMRI 381
- TraceSuspendThis, JVMRI 381
- tracing
 - Linux
 - ltrace tool 126
 - mtrace tool 126
 - strace tool 126
- tracing leaks, Windows 144
- tracing tools
 - Linux 126
- trailer record 1 in a heapdump 261
- trailer record 2 in a heapdump 261
- transaction dumps
 - z/OS 156
- tty, AIX 95
- type signatures 261
- Type, AIX 97
 - clnt 97
 - Description parameter 97
 - mmap 97
 - pers 97
 - work 97

U

- uid, AIX 95
- uname -a command, Linux 135
- understanding memory usage, AIX 105
- understanding the class loader 29

- UNKNOWN 196
- user exceptions, ORB 195
- user, AIX 95
- using dump agents 223
- utilities
 - NLS fonts
 - *.nix platforms 208

V

- verbose output
 - shared classes 363
- verboseHelper output
 - shared classes 364
- verboselO output
 - shared classes 363
- verification problems
 - shared classes 369
- versions, ORB 194
- virtual storage, z/OS 161
- vmstat command, Linux 135
- vmstat, AIX 99
- Vsid, AIX 97

W

- who should read this book xiv
- Windows
 - 32-bit large address aware
 - support 140
 - analyzing deadlocks 143
 - classifying leaks 144
 - collecting data 147
 - collecting data from a fault
 - condition 147
 - deadlocks 143
 - debugging performance problems
 - application profiling 146
 - finding the bottleneck 146
 - JIT compilation 146
 - JVM heap sizing 146
 - systems resource usage 146
 - diagnosing crashes 142
 - sending data to IBM 142
 - getting a dump from a hung
 - JVM 143
 - hangs 143
 - analyzing deadlocks 143
 - getting a dump 143
 - memory leaks 143
 - classifying leaks 144
 - memory model 144
 - tracing leaks 144
 - memory model 144
 - performance problems 145
 - problem determination 139, 140
 - sending data to IBM 142
 - setting up and checking
 - environment 139
 - system dump 141
 - tracing leaks 144
- work, AIX segment type 97

X

- Xcheckjni 445

Z

- z/OS
 - collecting data 164
 - crash diagnosis 153
 - crashes
 - documents to gather 153
 - failing function 154
 - dbx 152
 - debugging performance problems
 - application profiling 164
 - finding the bottleneck 163
 - JIT compilation 164
 - JVM heap sizing 164
 - systems resource usage 164
 - environment variables 149, 418
 - environment, checking 149
 - error message IDs 153
 - general debugging techniques 151
 - hangs 159
 - bad performance 161
 - IPCS commands 152
 - IPCS commands and sample
 - output 157
 - JVM dump initiation 242
 - LE settings 149
 - maintenance 149
 - memory leaks 161
 - LE HEAP 161
 - OutOfMemoryErrors 162
 - virtual storage 161
 - performance problems 163
 - private storage usage 149
 - process
 - deadlocked 160
 - looping 160
 - setting up dumps 150
 - TDUMPs 156



Printed in USA