

IBM® Developer Kit and Runtime Environment, Java™ 2  
Technology Edition, Version 5.0



# Diagnostics Guide



IBM® Developer Kit and Runtime Environment, Java™ 2  
Technology Edition, Version 5.0



# Diagnostics Guide

**Note**

Before using this information and the product it supports, read the information in Appendix F, "Notices," on page 379.

**Second Edition (March 2006)**

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 the first edition of this book and for this second edition are indicated by vertical bars to the left of the changes.

**© Copyright International Business Machines Corporation 2003, 2006. All rights reserved.**

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

---

# Contents

<b>Figures . . . . .</b>	<b>ix</b>	
<b>Tables . . . . .</b>	<b>xi</b>	
<b>About this book . . . . .</b>	<b>xiii</b>	
What does the "Java Virtual Machine (JVM)" mean?	xiii	
Who should read this book . . . . .	xiii	
Using this book . . . . .	xiii	
Other sources of information . . . . .	xiv	
Reporting problems in the JVM . . . . .	xiv	
Conventions and terminology used in this book . . . . .	xiv	
How to send your comments . . . . .	xv	
Contributors to this book . . . . .	xv	
Summary of changes . . . . .	xv	
For this second edition . . . . .	xv	
For the first edition . . . . .	xv	
<b>Part 1. Understanding the IBM Virtual Machine for Java . . . . .</b>	<b>1</b>	
<b>Chapter 1. The building blocks of the IBM Virtual Machine for Java. . . . .</b>	<b>3</b>	
Java application stack . . . . .	4	
IBM Virtual Machine for Java subcomponents . . . . .	4	
JVM API . . . . .	5	
Diagnostics component . . . . .	5	
Memory management . . . . .	5	
Class loader . . . . .	6	
Interpreter . . . . .	6	
Platform port layer . . . . .	6	
<b>Chapter 2. Understanding the Garbage Collector . . . . .</b>	<b>7</b>	
Overview of garbage collection . . . . .	7	
Object allocation . . . . .	7	
Reachable objects . . . . .	8	
Garbage collection . . . . .	8	
Heap size . . . . .	8	
Allocation . . . . .	9	
Heap lock allocation . . . . .	10	
Cache allocation . . . . .	10	
Large Object Area . . . . .	10	
Detailed description of garbage collection . . . . .	11	
Mark phase . . . . .	11	
Sweep phase . . . . .	14	
Compaction phase . . . . .	15	
Reference objects . . . . .	15	
Final reference processing . . . . .	16	
JNI weak reference . . . . .	16	
Heap expansion . . . . .	17	
Heap shrinkage . . . . .	17	
Generational Garbage Collector . . . . .	18	
Tenure age . . . . .	19	
Tilt ratio . . . . .	19	
How to do heap sizing . . . . .	20	
Initial and maximum heap sizes . . . . .	20	
Using verbose:gc . . . . .	21	
Using fine tuning options . . . . .	21	
Interaction of the Garbage Collector with applications . . . . .	21	
How to coexist with the Garbage Collector . . . . .	22	
Root set . . . . .	22	
Thread local heap . . . . .	22	
Bug reports . . . . .	22	
Finalizers . . . . .	23	
Manual invocation . . . . .	24	
Summary . . . . .	24	
Frequently asked questions about the Garbage Collector . . . . .	25	
<b>Chapter 3. Understanding the class loader . . . . .</b>	<b>29</b>	
The parent-delegation model . . . . .	29	
Name spaces and the runtime package . . . . .	30	
Why write your own class loader? . . . . .	30	
How to write your own class loader . . . . .	31	
<b>Chapter 4. Understanding shared classes . . . . .</b>	<b>33</b>	
<b>Chapter 5. Understanding the JIT . . . . .</b>	<b>35</b>	
JIT overview . . . . .	35	
How the JIT optimizes code . . . . .	36	
1) Inlining . . . . .	36	
2) Local optimizations . . . . .	36	
3) Control flow optimizations . . . . .	36	
4) Global optimizations . . . . .	37	
5) Native code generation . . . . .	37	
Frequently asked questions about the JIT . . . . .	37	
<b>Chapter 6. Understanding the ORB . . . . .</b>	<b>39</b>	
CORBA . . . . .	39	
RMI and RMI-IIOP . . . . .	39	
Java IDL or RMI-IIOP? . . . . .	40	
RMI-IIOP limitations . . . . .	40	
Further reading . . . . .	40	
Examples of client-server applications . . . . .	40	
Interfaces . . . . .	40	
Remote object implementation (or servant) . . . . .	41	
Stubs and ties generation . . . . .	41	
Server code . . . . .	42	
Summary of major differences between RMI (JRMP) and RMI-IIOP . . . . .	45	
Using the ORB . . . . .	46	
How the ORB works . . . . .	49	
The client side . . . . .	49	
The server side . . . . .	53	

## contents

Features of the ORB . . . . .	55	Escalating problem severity . . . . .	88
Portable object adapter . . . . .	55		
Fragmentation . . . . .	57	<b>Chapter 12. Submitting data with a problem report . . . . .</b>	<b>89</b>
Portable interceptors . . . . .	58	Sending files to IBM support . . . . .	89
Interoperable naming service (INS) . . . . .	60	Getting files from IBM support . . . . .	90
<b>Chapter 7. Understanding the Java Native Interface (JNI) . . . . .</b>	<b>63</b>	Using your own ftp server . . . . .	91
Overview of JNI . . . . .	63	Compressing files . . . . .	91
The JNI and the Garbage Collector . . . . .	64	When you will receive your fix . . . . .	91
Garbage Collector and object references . . . . .	64		
Garbage Collector and global references . . . . .	65		
Copying and pinning . . . . .	65		
Handling local references . . . . .	66	<b>Part 3. Problem determination . . . . .</b>	<b>93</b>
Local reference scope . . . . .	66		
Summary of local references . . . . .	68	<b>Chapter 13. First steps in problem determination . . . . .</b>	<b>95</b>
Local reference capacity . . . . .	68		
Manually handling local references . . . . .	68	<b>Chapter 14. AIX problem determination . . . . .</b>	<b>97</b>
Handling global references . . . . .	69	Setting up and checking your AIX environment . . . . .	97
Global reference capacity . . . . .	69	Enabling full AIX core files . . . . .	98
Handling exceptions . . . . .	69	General debugging techniques . . . . .	99
Using the isCopy flag . . . . .	69	Starting Javadumps in AIX . . . . .	99
Using the mode flag . . . . .	70	Starting Heapdumps in AIX . . . . .	99
A generic way to use the isCopy and mode flags . . . . .	71	AIX debugging commands . . . . .	99
Synchronization . . . . .	71	DBX Plug-in . . . . .	107
Debugging the JNI . . . . .	72	Diagnosing crashes . . . . .	108
JNI checklist . . . . .	73	Documents to gather . . . . .	108
<b>Chapter 8. Understanding Java Remote Method Invocation . . . . .</b>	<b>75</b>	Interpreting the stack trace . . . . .	108
The RMI implementation . . . . .	75	Debugging hangs . . . . .	109
Thread pooling for RMI connection handlers . . . . .	76	AIX deadlocks . . . . .	109
Understanding Distributed Garbage Collection (DGC) . . . . .	76	Investigating busy hangs in AIX . . . . .	110
Debugging applications involving RMI . . . . .	77	Poor performance on AIX . . . . .	112
<b>Part 2. Submitting problem reports . . . . .</b>	<b>79</b>	Understanding memory usage . . . . .	112
<b>Chapter 9. Overview of problem submission . . . . .</b>	<b>81</b>	32- and 64-bit JVMs . . . . .	112
How does IBM service Java? . . . . .	81	The 32-bit AIX Virtual Memory Model . . . . .	112
Submitting Java problem reports to IBM . . . . .	81	The 64-bit AIX Virtual Memory Model . . . . .	113
Java duty manager . . . . .	81	Changing the Memory Model (32-bit JVM) . . . . .	113
<b>Chapter 10. MustGather: Collecting the correct data to solve problems . . . . .</b>	<b>83</b>	The native and Java heaps . . . . .	114
Before you submit a problem report . . . . .	83	The AIX 32-bit JVM default memory models . . . . .	115
Data to include . . . . .	83	Monitoring the native heap . . . . .	115
Things to try . . . . .	84	Native heap usage . . . . .	116
Factors that affect JVM performance . . . . .	84	Specifying MALLOCTYPE . . . . .	117
Test cases . . . . .	84	Monitoring the Java heap . . . . .	117
Performance problems – questions to ask . . . . .	85	Receiving OutOfMemory errors . . . . .	117
<b>Chapter 11. Advice about problem submission . . . . .</b>	<b>87</b>	Is the Java or native heap exhausted? . . . . .	117
Raising a problem report . . . . .	87	Java heap exhaustion . . . . .	118
What goes into a problem report? . . . . .	87	Native heap exhaustion . . . . .	118
Problem severity ratings . . . . .	87	AIX fragmentation problems . . . . .	118
		Submitting a bug report . . . . .	119
		Debugging performance problems . . . . .	119
		Finding the bottleneck . . . . .	120
		CPU bottlenecks . . . . .	120
		Memory bottlenecks . . . . .	124
		I/O bottlenecks . . . . .	125
		JVM heap sizing . . . . .	125
		JIT compilation and performance . . . . .	125
		Application profiling . . . . .	125
		Collecting data from a fault condition in AIX . . . . .	125
		Getting AIX technical support . . . . .	126

<b>Chapter 15. Linux problem determination . . . . .</b>	<b>127</b>	Using HeapDump to debug memory leaks . . . . .	149
Setting up and checking your Linux environment	127	Debugging performance problems . . . . .	149
Working directory . . . . .	127	Finding the bottleneck . . . . .	149
Linux core files . . . . .	127	Windows systems resource usage . . . . .	149
Threading libraries . . . . .	128	JVM heap sizing . . . . .	150
General debugging techniques . . . . .	128	JIT compilation and performance . . . . .	150
Starting Javadumps in Linux . . . . .	128	Application profiling . . . . .	150
Starting heapdumps in Linux . . . . .	129	Collecting data from a fault condition in Windows	150
Using the dump extractor on Linux . . . . .	129		
Using core dumps . . . . .	129		
Using system logs . . . . .	129		
Linux debugging commands . . . . .	131		
Diagnosing crashes . . . . .	134		
Checking the system environment . . . . .	134		
Gathering process information . . . . .	134		
Finding out about the Java environment . . . . .	135		
Debugging hangs . . . . .	135		
Debugging memory leaks . . . . .	135		
Debugging performance problems . . . . .	136		
Finding the bottleneck . . . . .	136		
CPU usage . . . . .	136		
Memory usage . . . . .	137		
Network problems . . . . .	137		
JVM heap sizing . . . . .	138		
JIT compilation and performance . . . . .	138		
Application profiling . . . . .	138		
Collecting data from a fault condition in Linux . . . . .	138		
Collecting core files . . . . .	138		
Producing Javadumps . . . . .	138		
Using system logs . . . . .	138		
Determining the operating environment . . . . .	139		
Sending information to Java Support . . . . .	139		
Collecting additional diagnostic data . . . . .	139		
Known limitations on Linux . . . . .	140		
Threads as processes . . . . .	140		
Floating stacks limitations . . . . .	140		
glibc limitations . . . . .	140		
Font limitations . . . . .	140		
<b>Chapter 16. Windows problem determination . . . . .</b>	<b>143</b>		
Setting up and checking your Windows environment . . . . .	143		
Setting up your Windows environment for data collection . . . . .	144		
General debugging techniques . . . . .	145		
Starting Javadumps in Windows . . . . .	145		
Starting Heapdumps in Windows . . . . .	145		
Using the Cross-Platform Dump Formatter . . . . .	145		
System dump . . . . .	145		
Diagnosing crashes in Windows . . . . .	146		
Data to send to IBM . . . . .	147		
Debugging hangs . . . . .	147		
Analyzing deadlocks . . . . .	147		
Getting a dump from a hung JVM . . . . .	147		
Debugging memory leaks . . . . .	148		
The Windows memory model . . . . .	148		
Classifying leaks . . . . .	148		
Tracing leaks . . . . .	148		
		Using HeapDump to debug memory leaks . . . . .	149
		Debugging performance problems . . . . .	149
		Finding the bottleneck . . . . .	149
		Windows systems resource usage . . . . .	149
		JVM heap sizing . . . . .	150
		JIT compilation and performance . . . . .	150
		Application profiling . . . . .	150
		Collecting data from a fault condition in Windows	150
<b>Chapter 17. z/OS problem determination . . . . .</b>	<b>151</b>		
Setting up and checking your z/OS environment . . . . .	151		
Maintenance . . . . .	151		
LE settings . . . . .	151		
Environment variables . . . . .	151		
Private storage usage . . . . .	151		
Setting up dumps . . . . .	152		
General debugging techniques . . . . .	152		
Starting Javadumps in z/OS . . . . .	152		
Starting Heapdumps in z/OS . . . . .	152		
Using IPCS commands . . . . .	152		
Using dbx . . . . .	153		
Interpreting error message IDs . . . . .	153		
Diagnosing crashes . . . . .	153		
Documents to gather . . . . .	153		
Determining the failing function . . . . .	154		
Working with TDUMPs using IPCS . . . . .	155		
Debugging hangs . . . . .	160		
The process is deadlocked . . . . .	160		
The process is looping . . . . .	160		
The process is performing badly . . . . .	161		
Debugging memory leaks . . . . .	161		
Allocations to LE HEAP . . . . .	161		
z/OS virtual storage . . . . .	161		
OutOfMemoryErrors . . . . .	162		
Debugging performance problems . . . . .	163		
Finding the bottleneck . . . . .	163		
z/OS systems resource usage . . . . .	163		
JVM heap sizing . . . . .	163		
JIT compilation and performance . . . . .	164		
Application profiling . . . . .	164		
Collecting data from a fault condition in z/OS . . . . .	164		
<b>Chapter 18. Sun Solaris problem determination . . . . .</b>	<b>165</b>		
<b>Chapter 19. Hewlett-Packard SDK problem determination . . . . .</b>	<b>167</b>		
<b>Chapter 20. ORB problem determination . . . . .</b>	<b>169</b>		
Identifying an ORB problem . . . . .	169		
What the ORB component contains . . . . .	169		
What the ORB component does not contain . . . . .	170		
Platform-dependent problem . . . . .	170		
JIT problem . . . . .	170		
Fragmentation . . . . .	170		
Packaging . . . . .	170		
ORB versions . . . . .	170		

## contents

Limitation with bidirectional GIOP . . . . .	171
Debug properties . . . . .	171
ORB exceptions . . . . .	172
User exceptions . . . . .	172
System exceptions . . . . .	172
Completion status and minor codes . . . . .	173
Java security permissions for the ORB . . . . .	173
Interpreting the stack trace . . . . .	174
Description string . . . . .	174
Nested exceptions . . . . .	175
Interpreting ORB traces . . . . .	175
Message trace . . . . .	175
Comm traces . . . . .	176
Client or server . . . . .	177
Service contexts . . . . .	177
Common problems . . . . .	178
ORB application hangs . . . . .	178
Running the client without the server running before the client is invoked . . . . .	179
Client and server are running, but not naming service . . . . .	179
Running the client with MACHINE2 (client) unplugged from the network . . . . .	180
IBM ORB service: collecting data . . . . .	180
Preliminary tests . . . . .	180
Data to be collected . . . . .	181
<b>Chapter 21. NLS problem determination . . . . .</b>	<b>183</b>
Overview of fonts . . . . .	183
Font specification properties . . . . .	183
Fonts installed in the system . . . . .	183
Font utilities . . . . .	184
Font utilities on AIX, Linux, and z/OS . . . . .	184
Font utilities on Windows systems . . . . .	184
Common problems and possible causes . . . . .	184
<b>Part 4. Using diagnostic tools . . . . .</b>	<b>187</b>
<b>Chapter 22. Overview of the available diagnostics . . . . .</b>	<b>189</b>
Categorizing the problem . . . . .	189
Platforms . . . . .	189
Summary of diagnostic information . . . . .	190
Summary of cross-platform tooling . . . . .	191
Heapdump analysis tooling . . . . .	191
Cross-platform dump formatter . . . . .	191
JVMTI tools . . . . .	192
JVMPPI tools . . . . .	192
JPDA tools . . . . .	192
Trace formatting . . . . .	192
JVMRI . . . . .	193
<b>Chapter 23. Using dump agents . . . . .</b>	<b>195</b>
Help options . . . . .	195
Dump types and triggering . . . . .	197
Types of dump agents - examples . . . . .	197
Console dumps . . . . .	197
System dumps . . . . .	197
Tool option . . . . .	198
Javadumps . . . . .	198
Heapsdumps . . . . .	198
Snap traces . . . . .	199
Default dump agents . . . . .	199
Default settings for dumps . . . . .	200
Limiting dumps using filters and range keywords . . . . .	201
Removing dump agents . . . . .	201
Controlling dump ordering . . . . .	201
Controlling dump file names . . . . .	202
<b>Chapter 24. Using Javadump . . . . .</b>	<b>203</b>
Enabling a Javadump . . . . .	203
The location of the generated Javadump . . . . .	203
Triggering a Javadump . . . . .	204
Interpreting a Javadump . . . . .	204
Javadump tags . . . . .	205
Title, GPInfo, and EnvInfo sections . . . . .	205
Storage Management (MEMINFO) . . . . .	207
Locks, monitors, and deadlocks (LOCKS) . . . . .	207
Threads and stack trace (THREADS) . . . . .	208
Classloaders and Classes (CLASSES) . . . . .	209
Environment variables and Javadump . . . . .	210
<b>Chapter 25. Using Heapdump . . . . .</b>	<b>213</b>
Summary of Heapdump . . . . .	213
Information for users of previous releases of Heapdump . . . . .	213
Getting Heapdumps . . . . .	213
Enabling text formatted ("classic") Heapdumps . . . . .	214
Location of the generated Heapdump . . . . .	214
Producing a Heapdump using jdumpview . . . . .	214
Available tools for processing Heapdumps . . . . .	215
Using verbose:gc to obtain heap information . . . . .	215
Environment variables and Heapdump . . . . .	215
<b>Chapter 26. Using core (system) dumps . . . . .</b>	<b>217</b>
Overview . . . . .	217
Defaults . . . . .	217
Environment variables and core dumps . . . . .	218
Platform-specific variations . . . . .	219
z/OS . . . . .	219
Windows . . . . .	220
AIX and Linux . . . . .	220
<b>Chapter 27. Using method trace . . . . .</b>	<b>221</b>
Running with method trace . . . . .	221
Examples of use . . . . .	222
Where does the output appear? . . . . .	222
Advanced options . . . . .	222
Real example . . . . .	223
<b>Chapter 28. Using the dump formatter . . . . .</b>	<b>225</b>
What the dump formatter is . . . . .	225
Problems to tackle with the dump formatter . . . . .	226
Supported commands . . . . .	227
General commands . . . . .	227
Commands for analysing the memory . . . . .	229

Commands for working with classes . . . . .	230	-Xtgc:parallel . . . . .	262
Commands for working with objects . . . . .	230	-Xtgc:references . . . . .	262
Commands for working with Heapdumps . . . . .	231	-Xtgc:scavenger . . . . .	262
Commands for working with trace . . . . .	232	-Xtgc:terse . . . . .	263
Example session . . . . .	232	Native memory use by the JVM . . . . .	263
<b>Chapter 29. JIT problem determination</b>	<b>243</b>	Native code . . . . .	263
Disabling the JIT . . . . .	243	Large native objects . . . . .	264
Selectively disabling the JIT . . . . .	243		
Locating the failing method . . . . .	244		
Identifying JIT compilation failures . . . . .	245		
Performance of short-running applications . . . . .	246		
<b>Chapter 30. Garbage Collector diagnostics</b>	<b>247</b>		
How do the garbage collectors work? . . . . .	247		
Common causes of perceived leaks . . . . .	247		
Listeners . . . . .	247	Understanding shared classes diagnostics output . . . . .	267
Hash tables . . . . .	248	Verbose output . . . . .	267
Static data . . . . .	248	VerboseIO output . . . . .	267
JNI references . . . . .	248	VerboseHelper output . . . . .	268
Premature expectation . . . . .	248	printStats utility . . . . .	268
Objects with finalizers . . . . .	248	printAllStats utility . . . . .	269
Basic diagnostics (-verbose:gc) . . . . .	248	Deploying Shared Classes . . . . .	270
Garbage collection triggered by System.gc() .	249	Cache naming . . . . .	270
Allocation failures . . . . .	250	Cache housekeeping . . . . .	271
Global collections . . . . .	251	Cache performance . . . . .	271
Scavenger collections . . . . .	252	Dealing with runtime bytecode modification . . . . .	273
Concurrent garbage collection . . . . .	253	Potential problems with runtime bytecode modification . . . . .	273
Advanced diagnostics . . . . .	257	Modification contexts . . . . .	274
-Xdisableexplicitgc . . . . .	258	SharedClassHelper partitions . . . . .	274
-Xgcthreads . . . . .	258	Safemode . . . . .	274
-Xconcurrentbackground<number> . . . . .	258	Further considerations for runtime bytecode modification . . . . .	275
-Xconcurrentlevel<number> . . . . .	258	Understanding dynamic updates . . . . .	275
-Xgcworkpackets<number> . . . . .	258	Storing classes . . . . .	276
-Xclassgc . . . . .	258	Finding classes . . . . .	276
-Xalwaysclassgc . . . . .	258	Marking classes as stale . . . . .	277
-Xnoclassgc . . . . .	258	Redeeming stale classes . . . . .	277
-Xcompactgc . . . . .	259	Using the Java Helper API . . . . .	277
-Xnocompactgc . . . . .	259	General API Helper usage . . . . .	278
-Xcompactexplicitgc . . . . .	259	Debugging problems with shared classes . . . . .	279
-Xnocompactexplicitgc . . . . .	259	Using shared classes trace . . . . .	279
-Xpartialcompactgc . . . . .	259	Why classes in the cache might not be found or stored . . . . .	279
-Xnopartialcompactgc . . . . .	259	Dealing with initialization problems . . . . .	280
-Xenablestringconstantgc . . . . .	259	Dealing with verification problems . . . . .	281
-Xdisablestringconstantgc . . . . .	259	Dealing with cache problems . . . . .	282
-Xlp . . . . .	259	Class sharing with OSGi ClassLoading framework . . . . .	282
-Xnoloa . . . . .	259		
-Xloainitital<number> . . . . .	259		
-Xloamaximum<number> . . . . .	259		
-Xconmeter:<soa   loa   dynamic> . . . . .	259		
-Xenableexcessivevgc . . . . .	260		
-Xdisableexcessivevgc . . . . .	260		
-Xsoftrefthreshold<number> . . . . .	260		
-Xtgc tracing . . . . .	260		
-Xtgc:backtrace . . . . .	260		
-Xtgc:compaction . . . . .	260		
-Xtgc:concurrent . . . . .	261		
-Xtgc:dump . . . . .	261		
-Xtgc:freelist . . . . .	261		
<b>Chapter 31. Class-loader diagnostics</b>	<b>265</b>		
Class-loader command-line options . . . . .	265		
Class-loader runtime diagnostics . . . . .	265		
Loading from native code . . . . .	266		
<b>Chapter 32. Shared classes diagnostics</b>	<b>267</b>		
Understanding shared classes diagnostics output . . . . .	267		
Verbose output . . . . .	267		
VerboseIO output . . . . .	267		
VerboseHelper output . . . . .	268		
printStats utility . . . . .	268		
printAllStats utility . . . . .	269		
Deploying Shared Classes . . . . .	270		
Cache naming . . . . .	270		
Cache housekeeping . . . . .	271		
Cache performance . . . . .	271		
Dealing with runtime bytecode modification . . . . .	273		
Potential problems with runtime bytecode modification . . . . .	273		
Modification contexts . . . . .	274		
SharedClassHelper partitions . . . . .	274		
Safemode . . . . .	274		
Further considerations for runtime bytecode modification . . . . .	275		
Understanding dynamic updates . . . . .	275		
Storing classes . . . . .	276		
Finding classes . . . . .	276		
Marking classes as stale . . . . .	277		
Redeeming stale classes . . . . .	277		
Using the Java Helper API . . . . .	277		
General API Helper usage . . . . .	278		
Debugging problems with shared classes . . . . .	279		
Using shared classes trace . . . . .	279		
Why classes in the cache might not be found or stored . . . . .	279		
Dealing with initialization problems . . . . .	280		
Dealing with verification problems . . . . .	281		
Dealing with cache problems . . . . .	282		
Class sharing with OSGi ClassLoading framework . . . . .	282		
<b>Chapter 33. Tracing Java applications and the JVM</b>	<b>283</b>		
What can be traced? . . . . .	283		
Tracing methods . . . . .	283		
Tracing applications . . . . .	283		
Internal trace . . . . .	284		
Default tracing . . . . .	284		
Default Memory Management tracing . . . . .	284		
Where does the data go? . . . . .	285		
Placing trace data into in-storage buffers . . . . .	285		
Placing trace data into a file . . . . .	285		

## contents

External tracing . . . . .	286	RasInfo request types. . . . .	315
Tracing to stderr . . . . .	286	Intercepting trace data . . . . .	315
Trace combinations . . . . .	286	The -Xtrace:external=<option>. . . . .	315
Controlling the trace . . . . .	286	Calling external trace . . . . .	316
Specifying trace options . . . . .	286	Formatting . . . . .	316
Trace options summary . . . . .	287		
Detailed descriptions of trace options . . . . .	289		
Using the trace formatter . . . . .	298		
Trace properties file . . . . .	299		
What to trace . . . . .	299		
I Determining the tracepoint ID of a tracepoint . . . . .	299		
Application trace . . . . .	300		
Implementing application trace . . . . .	300		
<b>Chapter 34. Using the Reliability, Availability, and Serviceability Interface . . . . .</b>	<b>305</b>	<b>Part 5. Appendixes . . . . .</b>	<b>335</b>
Preparing to use JVMRI . . . . .	305	<b>Appendix A. CORBA minor codes . . . . .</b>	<b>337</b>
Writing an agent . . . . .	305	<b>Appendix B. Environment variables . . . . .</b>	<b>339</b>
Registering a trace listener . . . . .	306	Displaying the current environment . . . . .	339
Changing trace options . . . . .	307	Setting an environment variable . . . . .	339
Launching the agent . . . . .	307	Separating values in a list . . . . .	339
Building the agent. . . . .	307	JVM environment settings . . . . .	339
Agent design . . . . .	308	z/OS environment variables . . . . .	343
JVMRI functions . . . . .	308		
API calls provided by JVMRI . . . . .	308		
CreateThread . . . . .	308		
DumpDeregister . . . . .	309		
DumpRegister . . . . .	309		
DynamicVerbosegc . . . . .	309		
GenerateHeapdump . . . . .	310		
GenerateJavacore . . . . .	310		
GetComponentDataArea. . . . .	310		
GetRasInfo . . . . .	310		
InitiateSystemDump . . . . .	311		
InjectOutOfMemory . . . . .	311		
InjectSigSegv . . . . .	311		
NotifySignal. . . . .	311		
ReleaseRasInfo . . . . .	312		
RunDumpRoutine. . . . .	312		
SetOutOfMemoryHook . . . . .	312		
TraceDeregister. . . . .	312		
TraceRegister . . . . .	313		
TraceResume . . . . .	313		
TraceResumeThis . . . . .	313		
TraceSet . . . . .	313		
TraceSnap . . . . .	314		
TraceSuspend . . . . .	314		
TraceSuspendThis . . . . .	314		
RasInfo structure . . . . .	314		

---

## Figures

1.	The components of a typical Java Application Stack and the IBM JRE . . . . .	4
2.	Subcomponent structure of the IBM Virtual Machine for Java . . . . .	5
3.	New and old area in the generational concurrent heap . . . . .	18
4.	Example of heap layout before and after garbage collection for gencon . . . . .	19
5.	Tilt ratio . . . . .	20
6.	The ORB client side. . . . .	50
7.	Relationship between the ORB, the object adapter, the skeleton, and the object implementation . . . . .	56
8.	Simple portable object adapter architecture . . . . .	57
9.	Thread stack pointing to an object so that the Garbage Collector can see the object . . . . .	67
10.	Thread stack not pointing to an object so that the Garbage Collector cannot see the object . . . . .	67
11.	The AIX 32-Bit Memory Model with MAXDATA=0 (default) . . . . .	113
12.	Screenshot of the ReportEnv tool . . . . .	144
13.	Diagram of the DTFJ interface . . . . .	330



---

## Tables

1.	Commands for stubs and ties (skeletons)	41
2.	Stub and tie files . . . . .	42
3.	Deprecated Sun properties . . . . .	49
4.	JNI checklist . . . . .	73
5.	Packaging. . . . .	170
6.	Methods affected when running with Java 2 SecurityManager . . . . .	173
7.	Usage from java -Xdump:help . . . . .	195
8.	Types of dump . . . . .	196
9.	Keywords. . . . .	196
10.	Environment variables used to produce java dumps . . . . .	210
11.	Environment variables used to produce heap dumps . . . . .	215
12.	Signal mappings on different platforms . . . . .	219
13.	Options that control tracepoint selection . . . . .	287
14.	Options that indirectly affect tracepoint selection . . . . .	288
15.	Triggering and suspend or resume . . . . .	288
16.	Options that specify output files . . . . .	288
17.	MiscellaneousTrace control options . . . . .	288
18.	JVM environment settings — general options . . . . .	340
19.	Deprecated JIT options . . . . .	341
20.	Javadump and Heapdump options . . . . .	341
21.	Diagnostics options . . . . .	343
22.	Cross platform defaults . . . . .	377
23.	Platform-specific defaults . . . . .	378



---

## About this book

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

---

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

The installable Java package supplied by IBM comes in two versions:

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

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 Java Virtual Machine (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
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...)

For diagnostics information on other IBM SDKs, see <http://www-106.ibm.com/developerworks/java/jdk/diagnosis/>.

---

### 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 Java 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 13, "First steps in problem determination," on page 95. Some of the diagnostic tools described in this book do not apply to earlier versions.

You can use this book in three ways:

## Using this book

- As an overview of how the IBM Virtual Machine for Java operates, with emphasis on the interaction with Java. Part 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 and Part 3 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 of the book.

The parts overlap in some ways. For example, Part 3 refers to chapters that are in Part 4 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 if you read the appropriate chapter in Part 1.

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

---

## Other sources of information

- For the latest tools and documentation, see IBM developerWorks at:  
<http://www.ibm.com/developerworks/java/>
- For Java documentation, see:  
<http://java.sun.com/reference/docs/index.html>
- For the IBM Java SDKs, see IBM Java downloads at:  
<http://www.ibm.com/developerworks/java/jdk/index.html>

---

## Reporting problems in the JVM

If you want to use this book only to determine your problem and to send a problem report to IBM, go to Part 3, “Problem determination,” on page 93 of the book, and to the chapter that relates to your platform. Go to the section that describes the type of problem that you are having. This section might offer advice about how to correct the problem, and might also offer workarounds. The section will also tell you what data IBM service needs you to collect to diagnose the problem. Collect the data and send a problem report and associated data to IBM service, as described in Part 2, “Submitting problem reports,” on page 79 of the book.

---

## Conventions and terminology used in this book

Command-line options, system parameters, and class names are shown in bold. For example:

- **-Xgcthreads**
- **-Dibm.jvm.trusted.middleware.class.path**
- **java.security.SecureClassLoader**

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 book, 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 book, you can send them by e-mail to [jvmcookbook@uk.ibm.com](mailto:jvmcookbook@uk.ibm.com). Include the name of the book, the part number of the book, 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, a page number or table number).

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 79.

## Contributors to this book

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

## Summary of changes

### For this second edition

This second edition is based on its first edition, the *Diagnostics Guide* for Java 2 Technology Edition Version 5.0, SC34-6650-00. Technical changes made for this edition are indicated by revision bars to the left of the changes.

The significant changes for this second edition are:

- Improved information about debugging performance problems
- A new section, "Generational Garbage Collector" on page 18
- Reviews of the AIX, Linux, Windows, and z/OS chapters in Part 3, "Problem determination," on page 93
- Inclusion of Chapter 18, "Sun Solaris problem determination," on page 165
- Inclusion of Chapter 19, "Hewlett-Packard SDK problem determination," on page 167
- A new section, "Deploying Shared Classes" on page 270
- A new chapter, Chapter 37, "Using DTFJ," on page 327

### For the first edition

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. 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.

## contributors

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

You will find changes throughout the book, but the most significant changes are:

- New chapters:
  - Chapter 4, “Understanding shared classes,” on page 33
  - Chapter 14, “AIX problem determination,” on page 97, 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 32, “Shared classes diagnostics,” on page 267
  - Chapter 35, “Using the HPROF Profiler,” on page 319
  - Chapter 36, “Using the JVMTI,” on page 325
  - Appendix C, “Messages,” on page 345
- Much changed chapters:
  - Chapter 2, “Understanding the Garbage Collector,” on page 7
  - Chapter 22, “Overview of the available diagnostics,” on page 189
  - Chapter 24, “Using Javadump,” on page 203
  - Chapter 30, “Garbage Collector diagnostics,” on page 247

---

## Part 1. Understanding the IBM Virtual Machine for Java

The information in this part of the book will give you a basic understanding of the JVM. It 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 fairly large amount of information about the garbage collector is provided, because the garbage collector often seems to be the most difficult part of the JVM to understand.

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

This part also provides a chapter that describes the IBM Object Request Broker (ORB) component. 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 a tool and runtime component that provides distributed computing through the OMG-defined CORBA IIOP communication protocol. The ORB runtime consists of a Java implementation of a CORBA ORB. The ORB toolkit provides APIs and tools for both the RMI programming model and the IDL programming model.

The chapters in this part are:

- Chapter 1, “The building blocks of the IBM Virtual Machine for Java,” on page 3
- Chapter 2, “Understanding the Garbage Collector,” on page 7
- Chapter 3, “Understanding the class loader,” on page 29
- Chapter 4, “Understanding shared classes,” on page 33
- Chapter 5, “Understanding the JIT,” on page 35
- Chapter 6, “Understanding the ORB,” on page 39
- Chapter 7, “Understanding the Java Native Interface (JNI),” on page 63
- Chapter 8, “Understanding Java Remote Method Invocation,” on page 75



---

# 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 IBM Java Runtime Environment (JRE). The JVM is a virtualized computing machine that follows a well-defined specification for the runtime requirements of the Java programming language. It is called "virtual" because it provides a machine interface that is independent of 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 implementing concretely 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. However, not all compliant JVMs are identical. JVM implementers have a wide degree of freedom to define characteristics of the JVM 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 outputs platform-dependent machine-code. This compiler works in conjunction with the JVM to select parts of the Java program that would 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 compilation" (JIT). IBM's JIT compiler is described in Chapter 5, "Understanding the JIT," 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.

This *Diagnostics Guide* is not a JVM specification; it 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

### Java application stack

Figure 1 shows the components of a typical Java Application Stack and the IBM JRE.

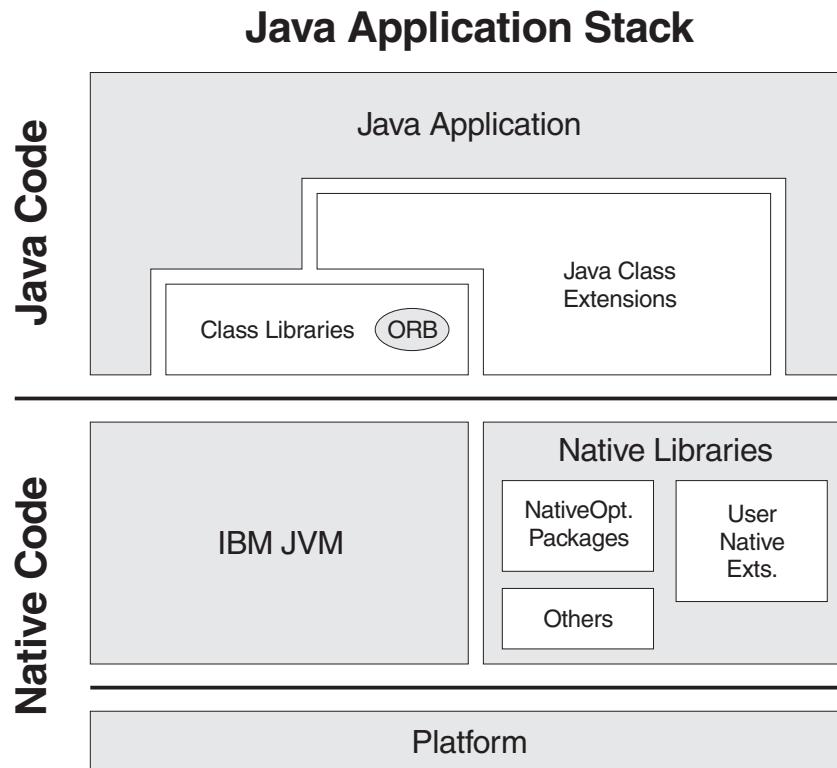


Figure 1. The components of a typical Java Application Stack and the IBM JRE

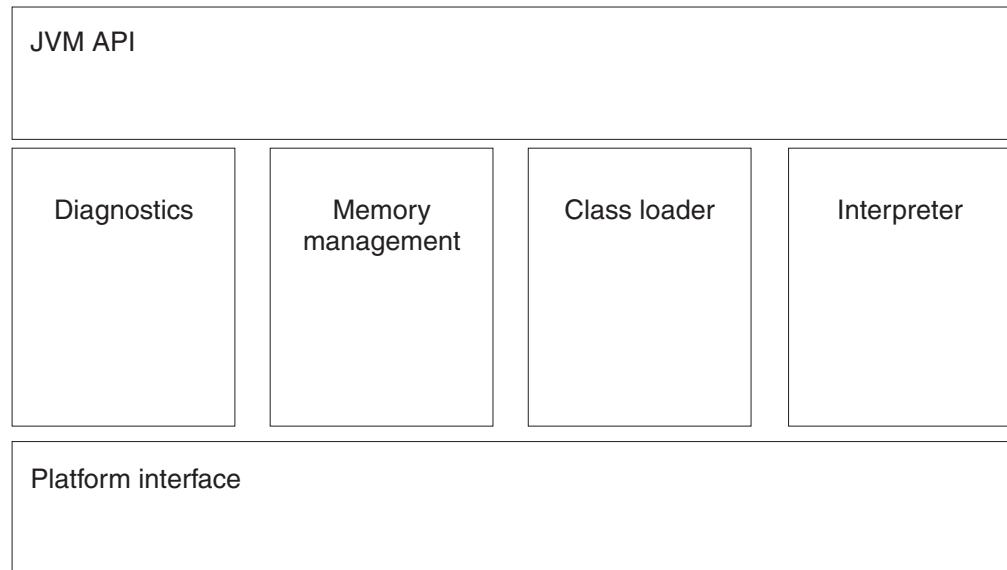
A Java application uses the Java class libraries that are provided by the JRE to implement the application-specific logic. The class libraries, in turn, are implemented in terms of other class libraries and, eventually, in terms of primitive native operations that are provided directly by the JVM. In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

The 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 7, “Understanding the Java Native Interface (JNI),” on page 63).

## IBM Virtual Machine for Java subcomponents

The IBM Virtual Machine for Java technology comprises a set of subcomponents that implement a logical grouping of function. Figure 2 shows the subcomponent structure of the IBM Virtual Machine for Java.

Figure 2 on page 5 shows subcomponent structure of the IBM Virtual Machine for Java.



*Figure 2. Subcomponent structure of the IBM Virtual Machine for Java*

## JVM API

The JVM API encapsulates all the interaction between external programs and the JVM. Examples 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 IBM Virtual Machine for Java 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, IBM's service engineers can use the capabilities of the diagnostics component to trace the runtime function of the JVM and 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 187 describes various uses of the diagnostics component.

## Memory management

The memory management subcomponent 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 subcomponent allocates the application a discrete region of unused memory. After the application no longer refers to the

## **Memory management**

storage, the memory management subcomponent 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 subcomponent has several policy options that you can specify when you deploy the application. Chapter 2, “Understanding the Garbage Collector,” on page 7 discusses memory management in the IBM Virtual Machine for Java.

## **Class loader**

The class loader subcomponent is responsible for supporting Java’s dynamic code loading facilities, including:

- 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, “Understanding the JIT,” on page 35.

## **Platform port layer**

The ability to reuse the code for the JVM across 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 subcomponents of the JVM are written in terms of the platform-neutral platform port layer functions. Further porting of the JVM requires the provision of concrete implementations of the platform port layer facilities rather than wholesale changes to the JVM code.

---

## Chapter 2. Understanding the Garbage Collector

This chapter describes the Garbage Collector under these headings:

- “Overview of garbage collection”
- “Allocation” on page 9
- “Detailed description of garbage collection” on page 11
- “Generational Garbage Collector” on page 18
- “How to do heap sizing” on page 20
- “Interaction of the Garbage Collector with applications” on page 21
- “How to coexist with the Garbage Collector” on page 22
- “Frequently asked questions about the Garbage Collector” on page 25

For detailed information about diagnosing Garbage Collector problems, see Chapter 30, “Garbage Collector diagnostics,” on page 247.

See also the reference information in “Garbage Collector command-line options” on page 370.

---

### Overview of garbage collection

This chapter provides:

- A summary of some of the diagnostic techniques that are described elsewhere in this book
- 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 active state of 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 from inside the JVM for storage for Java objects. Every allocation nominally 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, and that thread can allocate from its local heap area without the need to lock out other threads. This technique delivers the best possible allocation performance for small objects. Objects are allocated directly from a thread local allocation buffer, which the thread has previously allocated from the heap. A new object is allocated from this cache without the need to grab the heap lock. All objects less than 512 bytes (768 bytes on 64-bit platforms) 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

The active state of the JVM is made up of the set of stacks that represents the threads, the statics that are inside Java classes, and the set of local and global JNI references. All functions that are invoked inside the JVM itself cause a frame on the thread stack. This information is used to find the *roots*. 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 current heap because of lack of contiguous space, a memory allocation fault occurs, and the Garbage Collector is invoked. 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. It occurs in three phases:

- Mark
- Sweep
- Compaction (optional)

#### Mark phase

In the mark phase, all the objects that are referenced from the thread stacks, statics, interned strings, and JNI references are identified. This action creates the root set of objects that the JVM references. Each of those objects might, in turn, reference others. Therefore, the second part of the process is to scan each object for other references that it makes. These two processes together generate a bit vector that defines the beginning of all reachable objects.

#### Sweep phase

The sweep phase uses the mark bit vector generated by the mark phase to identify the chunks of heap storage that can be reclaimed for future allocations; these chunks are added to the pool of free space.

#### Compaction phase

When the garbage has been reclaimed from the heap, the Garbage Collector can consider compacting the resulting set of objects to remove the spaces that are between them. Because compaction can take a long time, the Garbage Collector only compacts when it is absolutely necessary. Compaction is, therefore, a rare event.

## Heap size

The maximum heap size is controlled by the `-Xmx` option. If this option is not specified, the default applies as follows:

| AIX 64 MB

Linux Half the real storage with a minimum of 16 MB and a maximum of 512 MB -1.

#### Windows

Half the real storage with a minimum of 16 MB and a maximum of 2 GB -1.

z/OS 64 MB.

The initial size of the heap is controlled by the `-Xms` option. If this option is not specified, the default applies as follows:

|     Windows, AIX, and Linux

|         4 MB

|     z/OS   1 MB

### Some basic heap sizing problems

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, but for best performance try to ensure that the heap never pages. The maximum heap size should, if possible, be able to be contained in physical memory to avoid paging.

**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 Garbage Collector is the JVM memory manager and is therefore 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”.

## Heap lock allocation

### Heap lock allocation

Heap lock allocation occurs when the allocation request cannot be satisfied in the existing cache; 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. If the Garbage Collector does not find enough free storage, it returns out of memory. The heap lock is released either after the object has been allocated, or if not enough free space is found.

### 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 platforms) 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 use of the TLH.

## Large Object Area

As objects are allocated and freed, the tenure area can become fragmented in such a way that allocation can be met only by time-consuming compactations. 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. The LOA is an area of the tenure area of the heap set used solely to satisfy allocations for large objects that cannot be satisfied in the main area of the tenure heap, which is referred to in the rest of this section as the small object area (SOA). 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, and z/OS only) but, if it is not used, it is shrunk to zero after a few collections. It can be disabled explicitly by specifying the `-Xnoloa` command-line option.

### Initialization and the LOA

The LOA boundary is calculated when the heap is initialized, and recalculated after every garbage collection. The initial size and maximum sizes of the LOA can be controlled by two command-line options: `-Xloainit` and `-Xloamax`. Both options take values between 0 and 0.95 (0% thru 95% of the current tenure heap size). If not specified, the default initial size of the LOA is 5% of the current tenure area and default maximum size is 50%.

### Expansion and shrinkage of the LOA

The Garbage Collector uses the following algorithm to expand or shrink the LOA, depending on usage:

- 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

When allocating an object, the allocation is first attempted in the 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 retried in the LOA. 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 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 is a stop-the-world (STW) operation, because all application threads are stopped while the garbage is collected.

### Mark phase

In this 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; an input packet from which references are popped and 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.

## Mark phase

- A bit vector called the *mark bit array* whose bits identify the objects that are reachable and have been visited. 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 is made up of the saved registers for each thread, the set of stacks that represent the threads, the statics that are in Java classes, and the set of local and global JNI references. All functions that are invoked 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 there are any references to unmarked objects, that is, mark bit is off, the object is marked by setting the appropriate bit in the mark bit array and the reference is pushed to the marking thread's output work packet. 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. If an overflow occurs, the Garbage Collector 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 slot 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 not degrade Mark performance on a uniprocessor, and to increase typical Mark performance on a multiprocessor system.

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. This thread performs very much as it always did, but 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 atomic primitives that require no additional lock.

By default, a platform with N processors also has N-1 new helper threads, that 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, there will be no helper threads. The **-Xgcthreads** option accepts any value greater than 0, but you gain little by setting it to more than N-1.

## Concurrent mark

Concurrent mark gives reduced and consistent garbage collection pause times when heap sizes increase. It starts a concurrent marking phase before the heap is full. In the concurrent phase, the Garbage Collector scans the roots, i.e. stacks, JNI references, class statics, etc. 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 Garbage Collector 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 activated every time a reference in an object is updated. The write barrier flags when an object reference update has occurred, to force a rescan of part of the heap. The heap is divided into 512-byte sections and each section is allocated a byte 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. An 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 Garbage Collector tries to start the concurrent mark phase so that it completes at the same time as the heap is exhausted. The Garbage Collector does this by constant tuning of the parameters that govern the concurrent mark time. In the STW phase, the Garbage Collector 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 overhead varies depending on how much idle CPU time is available for the background thread. Also, the write barrier has an overhead.

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

|     `-Xgcpolicy:<optthrput | optavgpause | gencon | subpool>`

Setting `-Xgcpolicy` to `optthrput` 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. `Optthrput` is the default setting. Setting `-Xgcpolicy` to `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. 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 disables concurrent mark. It uses an improved object allocation algorithm to achieve better performance

## Concurrent mark

when allocating objects on the heap. This option might improve performance on large SMP systems. The *subpool* option is available only on AIX, Linux PPC and zSeries, and z/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. To avoid filling this free space pool with lots of small chunks of storage, only chunks of at least a certain size are reclaimed and added to the free pool. The minimum size for a free chunk is currently defined as 512 bytes (768 bytes on 64-bit platforms).

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 zeroes that correspond to a length less than the minimum free size. When a sequence of sufficient length is found, the Garbage Collector 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 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 Garbage Collector 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 256KB and each thread (helper or master) takes a section at a time and scans it, performing a modified bit-wise 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 complete 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, as described previously. The concurrent sweep process is split into two types of operations:

- Sweep analysis, which processes sections of data in the mark map and determines ranges of free or potentially free memory to be added to the free list
- Connection, which takes analyzed sections of the heap and connects them into the free list

Any section of heap must first have been analyzed before it is connected into the free list. Heap sections are calculated in the same way as for parallel 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. Within 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 GC 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 be completed before a compaction can occur. Consequently, an STW collection that compacts will not have concurrent sweep active during the next round of execution.

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

## Compaction phase

When the garbage has been removed from the heap, the Garbage Collector can consider compacting the resulting set of objects, to remove the spaces that are between them. The process of compaction is complicated because if any object is moved, the Garbage Collector must change all the references that exist to it.

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 of the following are true and **-Xnocompactgc** has not been specified:

- **-Xcompactgc** has been specified.
- Following the sweep phase, not enough free space is available to satisfy the allocation request.
- A **System.gc()** has been requested and the last allocation failure 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 active heap is free.

## Reference objects

When a reference object is created, it is added to a list of reference objects of the same type. Instances of **SoftReference**, **WeakReference**, and **PhantomReference** are created by the user and cannot be changed; they cannot be made to refer to objects

## Reference objects

other than the object that they referenced on creation. Objects whose classes define a finalize method result in a pointer to that object being placed on 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

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. The referent is the object the reference object points to. 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 Out Of Memory exception 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 simply removed from the reference list.

## Final reference processing

The processing of objects that require finalization is more straightforward. The list of objects is processed and any element that is not marked is processed by marking and tracing the object and then creating an entry on the finalizable object list for the object. Then GC removes the element on the unfinalized object list. The final method for the object is then run at an undetermined point in the future by the reference handler thread.

## JNI weak reference

JNI weak references provide the same capability as WeakReference objects do, 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. Note that failure to delete a JNI Weak reference causes a memory leak in the table and performance problems. This is also true for 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 active part of the heap is expanded up to the maximum if one of the following is true:

- The Garbage Collector 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:

- If the heap is being expanded because less than **-Xminf** (default 30%) free space is available, the Garbage Collector calculates how much the heap needs to expand to get **-Xminf** free space.

If this is greater than the maximum expansion amount, which you can set with the **-Xmaxe** parameter (default of 0, which means no maximum expansion), the calculation is reduced to **-Xmaxe**.

If this is less than the minimum expansion amount, which you can set with the **-Xmine** parameter (default of 1 MB), it is increased to **-Xmine**.

- If the heap is expanding and the JVM is spending more than the maximum time threshold, the Garbage Collector calculates how much expansion is needed to expand the heap by 17% free space. This is adjusted as above, depending on **-Xmaxe** and **-Xmine**.
- Finally, the Garbage Collector ensures that the heap is expanded by at least the allocation request if garbage collection did not free enough storage.

All calculated expansion amounts are rounded up to a 512 byte boundary on 32-bit architecture, or a 1024 byte boundary on 64-bit architecture.

## Heap shrinkage

Heap shrinkage occurs after garbage collection while exclusive access of the virtual machine is still held. Shrinkage does not occur if any of the following are true:

- The Garbage Collector 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 Garbage Collector 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 architecture, or a 1024 byte boundary on 64-bit architecture.

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

## Heap expansion

- 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 Garbage Collector did not shrink and compact on the last garbage collection cycle.

Note that, 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*; that is, the ability of the native operating system to commit and decommit physical storage to the virtual storage, the Garbage Collector uses this function. In this case, physical memory can be decommitted on a heap shrinkage.

To summarize. You never see the amount of virtual memory 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.

Also note that, where paging is supported, the Garbage Collector 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 Garbage Collector

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

The Generational Garbage Collector has been introduced in IBM Java 5.0. A generational garbage collection strategy is well suited to an application that creates many short-lived objects, as is typical of many transactional applications. 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 live long enough, they are moved or promoted into the old area. Objects are promoted when they reach a certain age (known as the tenure age). This age is a count of the number of garbage collections for which they have lived.

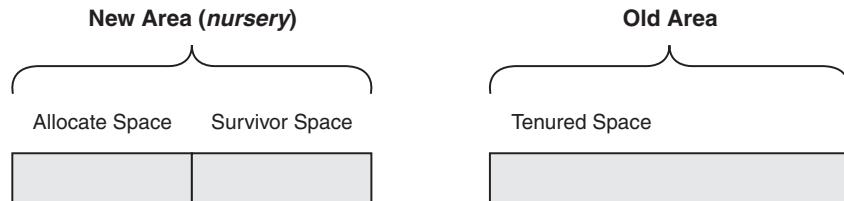


Figure 3. New and old area in the generational concurrent heap

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 GC process called scavenge is triggered. During a scavenge, live objects are copied either into the

survivor space or into the tenured space if they are old enough to have reached the tenured age. Dead objects in the nursery remain untouched. When all the live objects have been copied, the spaces in the new area switch roles. The new survivor space is now entirely empty of live objects and is available for the next scavenge.

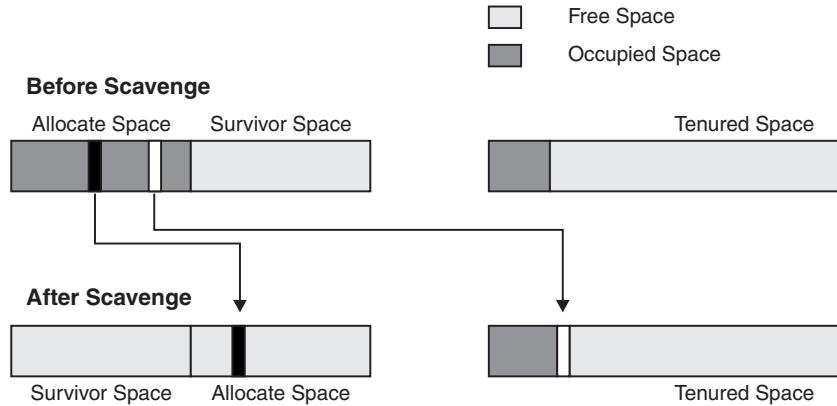


Figure 4. Example of heap layout before and after garbage collection for `gencon`

Figure 4 illustrates what happens during a scavenge. When the allocate space is full, a garbage collection is triggered. Live objects are then traced and copied into the survivor space. Objects that have reached the tenure age - that is, they have already been copied within the nursery a number of times - are promoted into the old area. 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 area 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  $x$  implies that, if the object has survived  $x$  flips between survivor and allocate space, it is promoted. 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 ratio can be adjusted to make the survivor space smaller. For example, if the initial total nursery 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. Figure 5 on page 20 illustrates how the boundary between allocate and survivor space is affected by the tilt ratio.

## Garbage collection - how to do heap sizing



Figure 5. Tilt ratio

## How to do heap sizing

This section describes how to 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

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, **-Xms 512M -Xmx 512M**. Using the same values is not usually a good idea, because it delays the start of garbage collection until the heap is full. The first time that the Garbage Collector runs, therefore, becomes a very expensive operation. Also, the heap is more likely to be fragmented and require a heap compaction. Again this is a very expensive operation. The recommendation is to start your application with the minimum heap size that it needs. When it starts up, the Garbage Collector will run often and, because the heap is small, efficiently.

The Garbage Collector takes these steps:

1. If the Garbage Collector finds enough garbage, it goes to step three.  
If it cannot find enough garbage, it goes to step two.
2. The Garbage Collector runs compaction.
3. If it cannot find enough garbage after compaction, or any of the other conditions for heap expansion are met (see “Heap expansion” on page 17) the Garbage Collector expands the heap.

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

From the above description, you can see that the Garbage Collector 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 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 overheads of expanding the heap are almost trivial compared to the cost of collecting and compacting a very large fragmented heap.

### Using verbose:gc

The verbose:gc output is fully described in Chapter 30, “Garbage Collector diagnostics,” on page 247. 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 377), 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 20.

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

Refer to 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.

#### -Xminfg and -Xmaxf

Minimum and maximum garbage collection time threshold.

These are also described in “Heap expansion” on page 17 and “Heap shrinkage” on page 17.

---

## Interaction of the Garbage Collector with applications

This interaction can be expressed as a contract between the Garbage Collector and an application. The Garbage Collector honors this contract:

1. The Garbage Collector will collect unreachable objects.
2. The Garbage Collector will not collect reachable objects
3. The Garbage Collector will stop all threads when it is running.
4. Garbage Collector invocation:
  - a. The Garbage Collector will not run itself except when an allocation failure occurs.
  - b. The Garbage Collector will honor manual invocations unless the **-Xdisableexplicitgc** parameter is specified.
5. The Garbage Collector will collect garbage at its own convenience, sequence, and timing, subject to clause 4b.
6. The Garbage Collector will honor all command-line variables, environment variables, or both.
7. Finalizers:
  - a. Are not run in any particular sequence
  - b. Are not run at any particular time
  - c. Are not guaranteed to run at all
  - d. Will run asynchronously to the Garbage Collector

## Interaction of the Garbage Collector with applications

This contract is used in the following section for some advice.

Note clause 4b on page 21. The specification says that a manual invocation of the Garbage Collector (for example, through the `System.gc()` call) suggests that a garbage collection cycle might be run. In fact, the call is interpreted as “Do a full garbage collection scan unless a garbage collection cycle is already executing, or explicit garbage collection is disabled by specifying `-Xdisableexplicitgc`.”

---

## How to coexist with the Garbage Collector

### Root set

Consider the root set. It is mainly a pseudo-random set of references from what happened to be in the stacks and registers of the JVM threads at the time that the Garbage Collector was invoked. This 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 clause 5 on page 21). This has significant consequences for finalizers (clause 7 on page 21), which are described more fully in “Finalizers” on page 23.

### Thread local heap

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 Garbage Collector 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 noncollectable, and allocated to a thread. The thread can now suballocate from the TLH, objects that are below a defined size. No heap lock is needed, so 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 are frequent underlying causes of bug reports. An example of a regular bug report to Java service of the hello-world variety is one in which a simple program allocates some object or objects, clears references to these objects, then initiates a garbage collection cycle. The objects are not seen as collected, usually because the application has attached a finalizer that reports when it is run.

It should be clear from the contract and the unpredictable nature of the Garbage Collector that more than one valid reason exists for this:

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

See clause 1 on page 21. True garbage is always found eventually, but it is not possible to predict when (clause 5 on page 21).

### Finalizers

The Java service team strongly recommends that applications avoid the use of finalizers as far as possible. The JVM specification states that finalizers should be used as an emergency clear-up of, for example, hardware resources. The service team recommends that this should be the only use of finalizers. They should not be used to clean up Java software resources or for closedown processing of transactions.

The reasons for this recommendation are partly in the nature of finalizers and how they are permanently linked to garbage collection, and partly in the contract that is described in “Interaction of the Garbage Collector with applications” on page 21. These topics are examined more closely in the following sections.

#### Nature of finalizers

The JVM specification says nothing about finalizers, except that they are final in nature. Nothing states when, how, or even whether a finalizer is run. The only rule is that if and when it is run, it is final.

Final, in terms of a finalizer, means that the object is known not to be in use any more. Clearly, this can happen only when the object is not reachable. Only the Garbage Collector can determine this. Therefore, when the Garbage Collector runs, it determines which are the unreachable objects that have a finalizer method. Normally, such objects would be collected, and the Garbage Collector would be able to satisfy the memory allocation fault. Finalized garbage, however, must have its finalizer run before it can be collected. Therefore, no finalized garbage can be collected in the cycle that actually finds it. Finalizers therefore make a garbage collection cycle longer (the cycle has to detect and process the objects) and less productive. Finalizers are an overhead on garbage collection. Because garbage collection is a stop-the-world operation, it makes sense to reduce this overhead as far as possible.

Note that the Garbage Collector cannot run finalizers itself when it finds them. This is because a finalizer might run an operation that takes a long time, and the Garbage Collector cannot risk locking out the application while this operation is running. So finalizers must be collected into a separate thread for processing. This adds more overhead into the garbage collection cycle.

#### Finalizers and the garbage collection contract

Garbage Collector contract clause 7 on page 21, which shows the nonpredictable behavior of the Garbage Collector, has particular significant results:

- Because the graph of objects that the Garbage Collector finds is basically random, the sequence in which finalized objects are located has no relationship to the sequence in which they were created nor to the sequence in which their objects became garbage (contract subclause 7a on page 21). Similarly, the sequence in which finalizers are run is also random.
- Because the Garbage Collector has no knowledge of what is in a finalizer, or how many finalizers exist, it tries to satisfy an allocation without needing to process finalizers. If a garbage collection cycle cannot produce enough normal garbage, it might decide to process finalized objects. So it is not possible to predict when a finalizer is run (contract subclause 7b on page 21).
- Because a finalized object might be retained garbage, it is possible that a finalizer might not run at all (contract subclause 7c on page 21).

## How to coexist with the Garbage Collector

### How finalizers are run

If and when the Garbage Collector decides to process unreachable finalized objects, those objects are placed onto a queue that is input to a separate finalizer thread. When the Garbage Collector has ended and the threads are unblocked, this thread starts to perform its function. 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 contract subclause 7d on page 21. Of course, if running with a large heap, the next garbage collection cycle might not happen for quite a long time.

### Summary

- Finalizers are an expensive overhead.
- Finalizers are not dependable.

The Java service team would recommend that :

- Finalizers are not used for process control
- Finalizers are not used for tidying Java resources
- Finalizers are not used at all as far as possible

For tidying Java resources, think about the use of a clean up routine. When you have finished with an object, call the routine to null out all references, deregister listeners, clear out hash tables, and so on. This 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.

### Manual invocation

The Garbage Collector contract subclause 4b on page 21 notes that the Garbage Collector can honor a manual invocation; for example, through the `System.gc()` call. This call nearly always invokes a garbage collection cycle, which is expensive.

The Java service team recommend that this call is not used, or if it is, it is enveloped in conditional statements that block its use in an application runtime environment. The Garbage Collector is carefully adjusted to deliver maximum performance to the JVM. Forcing it to run severely degrades JVM performance.

From the previous sections, you can see that it is pointless trying to force the Garbage Collector to do something predictable, such as collecting your new garbage or running a finalizer. It might happen; it might not. Let the Garbage Collector run in the parameters that an application selects at start-up time. This method nearly always produces best performance.

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

### Summary

Do not try to control the Garbage Collector or to predict what will happen in a given garbage collection cycle. You cannot do it. This unpredictability is handled, and the Garbage Collector is designed to run well and efficiently inside these conditions. Set up the initial conditions that you want and let the Garbage

Collector run. It will honor the contract (described in “Interaction of the Garbage Collector with applications” on page 21), which is within the JVM specification.

## Frequently asked questions about the Garbage Collector

### What are the default heap sizes?

See “Heap size” on page 8.

### If I don't specify -Xmx and -Xms, what values will Java use?

See Appendix E, “Default settings for the JVM,” on page 377.

### What are default values for the native stack (-Xms0) and Java stack (-Xss)?

See Appendix E, “Default settings for the JVM,” on page 377.

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

**optthruput** disables concurrent mark. If you do not have pause time problems (indicated by erratic application response times), you should 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.

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

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

### How many GC helper threads are spawned? What is their work?

A platform with n processors will have 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.

### How can I prevent Java heap fragmentation?

Note that the following suggestions might not help avoid fragmentation in all cases.

- Start with a small heap. Set **-Xms** far lower than **-Xmx**. It might be appropriate to allow **-Xms** to default, because the default is a low value.
- Increase the maximum heap size, **-Xmx**.

### What is Mark Stack Overflow? 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 so is finite and can overflow. This situation is called Mark Stack Overflow (MSO). The algorithms to handle this situation are very expensive in processing terms, and so MSO is a big hit on GC performance.

### How can I prevent Mark Stack Overflow?

The following suggestions are not guaranteed to avoid MSO:

## Frequently asked questions about the Garbage Collector

- 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.

- In the event of MSO, you will see entries in the verbose gc as follows:

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

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 an application's allocation requests 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, or 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 a lot of free heap storage, and releasing some heap memory is beneficial for system performance. Heap shrinkage occurs after GC, but when all the threads are still suspended.

### Does the IBM GC guarantee that it will clear all the unreachable objects?

The IBM GC guarantees only that all the objects that were not reachable at the beginning of the mark phase will be collected. While running concurrently, our GC guarantees only that all the objects that were unreachable when concurrent mark began will be 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 could occur because of

- 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 can be disabled using the option **-Xdisableexcessivegc**.

### When I see an OutOfMemoryError, does that mean that the Java program will exit?

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.

### How do I figure out if the Java heap is fragmented?

When you see (from `verbose:gc`) that the Java heap has a lot of free space, but the allocation request still fails, it usually points to a fragmented heap.

## Frequently asked questions about the Garbage Collector

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

You see this 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. So you might see more than one GC cycle for one allocation failure.

## Frequently asked questions about the Garbage Collector

---

## Chapter 3. Understanding the class loader

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.

The class loader loads, verifies, prepares and resolves, and initializes a class from a JVM class file.

- **Loading** involves obtaining the byte array representing the Java class file.
- **Verification** of a JVM 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 execution 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 within a JVM 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.

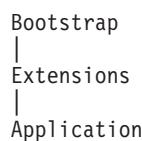
The delay is generally to increase execution speed. Not all symbols in a class file are referenced during execution. So, by delaying resolution, fewer symbols might have to be resolved, giving you less runtime overhead. Additionally, the cost of resolution is gradually reduced over the total execution 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 normally the repositories are less trusted:



## Class loader - parent-delegation model

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.

---

## Name spaces 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 name spaces that the class loader identifies.

A name space 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 name space, it is impossible to load another class of the same name into that name space. Multiple copies of any given class can be loaded because a name space is created for each class loader.

Name spaces 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 name spaces 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.

---

## Why write your own class loader?

The three main reasons for wanting to create your own class loader are:

- 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.

- To partition user code.

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

- 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.

---

## How to write your own class loader

Under the Java 1 class loading system, it was a requirement that any custom class loader must subclass `java.lang.ClassLoader` and override the abstract `loadClass()` method that was in the `ClassLoader`. The `loadClass()` method had to meet several requirements so that it could work effectively with the JVM's class loading mechanism, such as:

- Checking whether the class has previously been loaded
- Checking whether the class had been loaded by the system class loader
- Loading the class
- Defining the class
- Resolving the class
- Returning the class to the caller

The Java 2 class loading system has simplified the process for creating custom class loaders. The `ClassLoader` class was given a new constructor that takes the parent class loader as a parameter. This parent class loader can be either the application class loader, or another user-defined class loader. This allows any user-defined class loader to be contained easily into the delegation model.

Under the delegation model, the `loadClass()` method is no longer abstract, and as such does not need to be overridden. The `loadClass()` method handles the delegation class loader mechanism and should not be overridden, although it is possible to do so, so that Java 1 style `ClassLoaders` can run on a Java 2 JVM.

Because the delegation code is handled in `loadClass()`, in addition to the other requirements that were made of Java 1 custom class loaders, custom class loaders should override only the new `findClass()` method, in which the code to access the new class repository should be placed. The `findClass()` method is responsible only for loading the class bytes and returning a defined class. The method `defineClass()` can be used to convert class bytes into a Java class:

```
class NetworkClassLoader extends ClassLoader {  
    String host;  
    int port;  
  
    public Class findClass(String name) {  
        byte[] b = loadClassData(name);  
        return defineClass(name, b, 0, b.length);  
    }  
    private byte[] loadClassData(String name) {  
        // Load the class data from the connection  
    }  
}
```

## **How to write your own class loader**

## Chapter 4. Understanding shared classes

In the past, many JVMs have offered some form of class sharing; for example, the IBM Persistent Reusable JVM on z/OS and Sun Microsystems "CDS" feature in their Java 5.0 release. Class sharing in the IBM Version 5.0 SDK offers a completely transparent and dynamic means of sharing all loaded classes, both application classes and system classes, and placing no restrictions on JVMs that are sharing the class data (unless runtime bytecode modification is being used).

Key points to note about the IBM class sharing feature are as follows.

- Class sharing is available on every platform supported by IBM Java 5.0.
- Classes are stored in a named "class cache", which is an area of shared memory of fixed size, 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.
- When a cache becomes full, classes in the cache can still be shared, but no new classes 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 filesystem, 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.
- Sharing of bytecode that is modified at runtime is supported, but must be used with care.
- Access to the class 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'd code, and similar items cannot be stored in the cache.

Sharing all fixed class data for an application between multiple JVMs has obvious benefits:

- Firstly, the virtual memory footprint reduction when using more than one JVM instance can be significant.
- Additionally, 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.

## **Understanding Shared Classes**

---

## Chapter 5. Understanding the JIT

The Just-In-Time compiler (JIT) is not part of the JVM, but is a basic component of the SDK. This chapter summarizes the relationship between the JVM and the JIT and gives a short description of how the JIT works.

### JIT overview

Java is an interpreted language, so it has a Write Once Run Anywhere (WORA) capability. The Java compiler reads Java source files and outputs strings of *bytecodes*, which are platform-neutral pseudo-machine code. At runtime, the JVM reads these bytecodes, interprets the semantics of each individual bytecode, and performs the appropriate computation. This means that a JVM that is interpreting bytecodes has a slower performance than a native application consisting of machine code generated by a native compiler.

The JIT is therefore important because it helps improve the performance of Java applications. The JIT comes into use whenever a Java method is called; it compiles the bytecodes of that method into native machine code, compiling it "just in time" to execute. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it. However, when the JVM starts, thousands of methods are executed. They could incur a significant startup overhead, because of the time it takes the JIT to run and compile every method. This means that if a Java program is run without the JIT, the JVM starts up quickly but runs relatively slowly. Conversely, if the program is run with the JIT, the JVM starts up slowly, then runs quickly. In some applications, you might find that it takes longer to start the JVM than to run the application itself.

In practice, not all methods are compiled the first time they are called. For each method, the JVM maintains a call count, which is incremented every time the method is invoked. The JVM interprets a method until its call count exceeds a *JIT threshold*. Therefore, often-used methods are compiled soon after the JVM has started; conversely, the methods that are used less often are compiled much later or perhaps not at all. In fact, the compilation of methods is spread out over the life of the JVM. This way, the JVM starts up quickly, but the program can still have improved performance, because methods are compiled to native code when their call counts reach the JIT threshold. The threshold has been carefully selected to obtain an optimal balance between startup times and run-time performance.

After a method is compiled, its call count is reset to zero; subsequent calls to the method continue to increment its count. When the call count of a method reaches a *JIT recompilation threshold*, the JIT compiles it a second time, this time applying a larger selection of optimizations than on the previous compilation (because the method has proven to be a significant part of the whole program). The recompilation process is iterative; the call count of a recompiled method is reset again and, as it reaches succeeding thresholds, triggers recompilations at increasing optimization levels. Thus, the busiest methods of a Java program are always optimized most aggressively, maximizing the performance benefits of using the JIT. The JIT can also measure operational data at run time, and use that data to improve the quality of further recompilations.

### How the JIT optimizes code

When a method is chosen for compilation, the JVM feeds its bytecodes to the JIT. The JIT needs to understand the semantics and syntax of the bytecodes before it can compile the method correctly. To help the JIT 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 chapter provides a brief overview of the phases of JIT compilation. For more information, see Chapter 29, “JIT problem determination,” on page 243.

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.

#### 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

#### 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.

#### 3) Control flow optimizations

The following optimizations analyze the flow of control within a method (or specific sections of it) and rearrange code paths to improve their efficiency:

- 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

## 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:

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

## 5) Native code generation

The process of native code generation varies, 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 within the code cache is recorded, so that future calls to it will invoke the compiled code. At any given time, the JVM process consists of the JVM executables and a set of JIT-compiled code that is linked dynamically to the bytecode interpreter in the JVM.

Therefore, if a Java program crashes or hangs in the JVM process space, but outside the range of the JVM executable code in that process, the problem is likely to be within the code cache.

## Frequently asked questions about the JIT

### Can I disable the JIT?

Yes. Set the appropriate command-line parameter (see Appendix D, "Command-line options," on page 361). Alternatively, delete or rename the JIT library, which is located with the JVM executables and called j9jit23.dll (on Windows) or libj9jit23.so (on other platforms).

### Can I use another vendor's JIT?

No.

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

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

### Can I cancel the compiled code of a method and run it interpreted?

No. After a method has been compiled, it will not run interpreted during future invocations of the method. An exception is a method that is owned by a class that was loaded with a custom (user-written) class loader and that has since been unloaded.

### Can the JIT "decompile" methods?

That is, can compiled code be canceled? No.

### Can I control the JIT compilation?

Yes. See Chapter 29, "JIT problem determination," on page 243. Advanced diagnostics are available to IBM engineers.

## Frequently asked questions about the JIT

### Can I dynamically control the JIT?

No. You can set JIT parameters only at JVM startup time. The JIT can be started up only at the same time as the JVM.

---

## Chapter 6. Understanding the ORB

This chapter describes the Object Request Broker (ORB). The topics are:

- “CORBA”
- “RMI and RMI-IIOP”
- “Java IDL or RMI-IIOP?” on page 40
- “RMI-IIOP limitations” on page 40
- “Further reading” on page 40
- “Examples of client–server applications” on page 40
- “Using the ORB” on page 46
- “How the ORB works” on page 49
- “Features of the ORB” on page 55

---

### CORBA

Common Object Request Broker Architecture (CORBA) is an open, vendor-independent specification for distributed computing. It is published by the Object Management Group (OMG). Using the Internet Inter-ORB Protocol (IIOP), it allows objects on different architectures, operating systems, and networks to interoperate. This interoperability is obtained by the use of the Interface Definition Language (IDL), which specifies the syntax that is used to invoke operations on objects. IDL is programming-language independent.

Developers define the hierarchy, attributes, and operations of objects in IDL, 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.

OMG have produced specifications for mappings from IDL to many common programming languages, including C, C++, and Java. Central to the CORBA specification is the Object Request Broker (ORB). The ORB routes requests from client to remote object, and responses to their destinations. Java contains an implementation of the ORB that communicates by using IIOP.

---

### RMI and RMI-IIOP

RMI is Java’s traditional form of remote communication. Basically, 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) however, and the two protocols can communicate.

## RMI and RMI-IIOP

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?

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

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

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/rmi-iiop/rmi\_iiop\_pg.html. Example programs are provided in demo/rmi-iiop.

---

## Examples of client–server applications

Here, CORBA, RMI (JRMP), and RMI-IIOP approaches are going to be used to present three client-server hello-world applications. All the applications exploit the RMI-IIOP IBM ORB.

### Interfaces

These are the interfaces that are to be implemented:

- CORBA IDL Interface (Foo.idl):  

```
interface Foo { string message(); };
```
- JAVA RMI Interface (Foo.java):  

```
public interface Foo 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**, that does not need any parameter, and it returns a string. For further information about IDL and its mapping to Java see, the OMG specifications ([www.omg.org](http://www.omg.org)).

## Remote object implementation (or servant)

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

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

In the early versions of Java RMI (JRMP), the servant class had to extend the **java.rmi.server.UnicastRemoteObject** class. Now, you can use the class **PortableRemoteObject** for both RMI over JRMP and IIOP, thereby making the development of the remote object virtually independent of the protocol that is used. Also, the object implementation does not need to extend **PortableRemoteObject**, especially if it already extends another class (single-class inheritance). However, in this case, the remote object instance must be exported in the server implementation (see below). By exporting a remote object, you make that 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 FooImpl extends _FooPOA {
    public String message() { return "Hello World"; }
}
```

This implementation conforms to the Inheritance model in which the servant extends directly the IDL-generated skeleton **FooPOA**. 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 **FooOperations**). However, the Tie model introduces a level of indirection; 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 you can decide whether to use the Tie or the Delegate model. In RMI-IIOP however, 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. Table 1 shows what command should be run to get stubs and ties (or skeletons) for the three techniques.

*Table 1. Commands for stubs and ties (skeletons)*

CORBA	RMI(JRMP)	RMI-IIOP
idlj Foo.idl	rmic FooImpl	rmic -iiop Foo

The compilation generates the files that are shown in Table 2 on page 42. (Use the **-keep** option with **rmic** if you want to keep the intermediate .java files).

## ORB - examples of client-server applications

Table 2. Stub and tie files

CORBA	RMI(JRMP)	RMI-IIOP
Foo.java	FooImpl_Skel.class	_FooImpl_Tie.class
FooHolder.java	FooImpl_Stub.class	_Foo_Stub.class
FooHelper.java	Foo.class (Foo.java present)	Foo.class (Foo.java present)
FooOperations.java	FooImpl.class (only compiled)	FooImpl.class (only compiled)
_FooStub.java		
FooPOA.java (-fserver, -fall, -fserverTie, -fallTie)		
FooPOATie.java (-fserverTie, -fallTie)		
_FooImplBase.java (-oldImplBase)		

Since the J2SE v.1.4 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 still generate older versions of the server-side skeletons that are compatible with servers that are written in J2SE 1.3 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 that 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 CORBA environment or the RMI registry for a RMI (JRMP) application. Therefore, you can use JNDI in CORBA and in RMI cases, thereby making 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("foo", fooReference); // bind the reference to the name "foo"
Object obj = ctx.lookup("foo"); // obtain the reference
```

However, 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 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 hashtable as an argument to the constructor of InitialContext.

For example, with RMI(JRMP), you do not need to do much other than create an instance of the servant and 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 usually 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  
            FooImpl foo = new FooImpl();  
            foo.setORB(orb);  
  
            // TIE model ONLY  
            // create a tie, with servant being the delegate and  
            // obtain the reference ref for the tie  
            FooPOATie tie = new FooPOATie(foo, poa);  
            Foo ref = tie._this(orb);  
  
            // Inheritance model ONLY  
            // get object reference from the servant  
            org.omg.CORBA.Object ref = poa.servant_to_reference(foo);  
            Foo ref = FooHelper.narrow(ref);  
  
            // bind the object reference ref to the naming service using JNDI  
            .....(see previous code) .....
```

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
```

## ORB - examples of client-server applications

```
POA poa = (POA)orb.resolve_initial_references("RootPOA");
poa.the_POAManager().activate();

// Create servant and its tie
FooImpl foo = new FooImpl();
_FooImpl_Tie tie = (_FooImpl_Tie)Util.getTie(foo);

// 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 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(IOP) server code can be reduced to instantiating the servant (FooImpl foo = new FooImpl()) and binding it to a naming service as is usually 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.

You must remember also one more important fact when you decide between the JRMP and IIOP protocols. 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 FooClient {
    public static void main(String [] args) {
        try{
            Foo fooref
            //Look-up the naming service using JNDI and get the reference
            .....
            // Invoke method
            System.out.println(fooRef.message());
        }
        catch(Exception e) {}
    }
}
```

CORBA Client example:

```
public class FooClient {
    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
            Foo fooRef = FooHelper.narrow(o);
            // Method Invocation
            System.out.println(fooRef.message());
        }
        catch(Exception e) {}
    }
}
```

RMI-IIOP Client example:

```

public class FooClient {
    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
            Foo fooRef = (Foo)PortableRemoteObject.narrow(o, Foo.class);
            // Method Invocation
            System.out.println(fooRef.message());
        }
        catch(Exception e) {}
    }
}

```

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

This section examines the 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**

- Import statement:

```
import javax.rmi.PortableRemoteObject;
```

- Implementation class of a remote object:

```
public class FooImpl extends PortableRemoteObject implements Foo
```

- Name registration of a remote object:

```
NamingContext.rebind("Foo",ObjRef);
```

- Generate a tie for IIOP with rmic **-iiop**
- Run tnameserv as a name server
- Generate IDL with rmic **-idl** for CORBA clients

### **Summary of differences in client development**

- Import statement:

```
import javax.rmi.PortableRemoteObject;
```

- Identify a remote object by name:

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

```
MyObject myobj = (MyObject)PortableRemoteObject.narrow(obj,MyObject.class);
```

## ORB - examples of client-server applications

- Generate a stub for IIOP with rmic -iiop

## Using the ORB

To use the ORB, you need to understand the properties that the ORB contains. These properties change the behavior of the ORB as described in this section. 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 and thereby interrupt that thread's wait 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 complete.

- **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 a host name or the IP address (for example, 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 hostname is the name of the machine on which the initial server contact for this client resides.

**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 machine 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 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 1 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 machine 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.

## Using the ORB

- **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 PortableInterceptors are driven. This should improve performance if interceptors are not required when invoking a co-located object.
- **com.ibm.CORBA.ORBCharEncoding:** (default: ISO8859\_1)  
Specifies the ORB's native encoding set for character data.
- **com.ibm.CORBA.ORBWCharDefault:** (default: UCS2 )  
Indicates that wchar codeset UCS2 is to be used with other ORBs that do not publish a wchar codeset.
- **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 will 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 from that ORB's response.
- **com.ibm.CORBA.ServerSocketQueueDepth:** (range: 50 through 2147483647 ) (default: 0)  
The maximum queue length for incoming connection indications (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 output 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 within its address space (for example, the server containing the ORB). Its value can be any String; the default is a randomly generated number that is unique within the ORB's JVM.

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

Identifies the set of endpoints (hostname or IP address and port) on which the ORB will listen for requests. 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, but note that some versions of the ORB support only the first endpoint.

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, but it can be set to the ORB's default hostname, which 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.

Table 3 shows the Sun properties that are now deprecated and the IBM properties that have replaced them. These properties are not OMG standard properties, despite their names.

*Table 3. Deprecated Sun properties*

Sun property	IBM property
com.sun.CORBA.ORBServerHost	com.ibm.CORBA.LocalHost
com.sun.CORBA.ORBServerPort	com.ibm.CORBA.ListenerPort
org.omg.CORBA.ORBIInitialHost	com.ibm.CORBA.BootstrapHost
org.omg.CORBA.ORBIInitialPort	com.ibm.CORBA.BootstrapPort
org.omg.CORBA.ORBIInitialServices	com.ibm.CORBA.InitialReferencesURL

---

## How the ORB works

This section describes a simple, typical RMI-IIOP session in which a client accesses a remote object on a server by implementing an interface named *Foo*, and invokes a simple method called `message()`. This method returns a "Hello World" string. (See the examples that are given earlier in this chapter.)

Firstly, this section explains the client side, and describes what the ORB does under the cover and transparently to the client. Then, the important role of the ORB in the server side is explained.

### The client side

The subjects discussed here are:

- "Stub creation"
- "ORB initialization" on page 50
- "Getting hold of the remote object" on page 51
- "Remote method invocation" on page 52

#### Stub creation

In a simple distributed application, the client needs to know (in almost all the cases) what kind of object it is going to contact and which method of this object it

## How the ORB works

needs to invoke. Because the ORB is a general framework you must give it general information about the method that you want to invoke.

For this reason, you implement a Java interface, Foo, which contains the signatures of the methods that can be invoked in the remote object (see Figure 6).

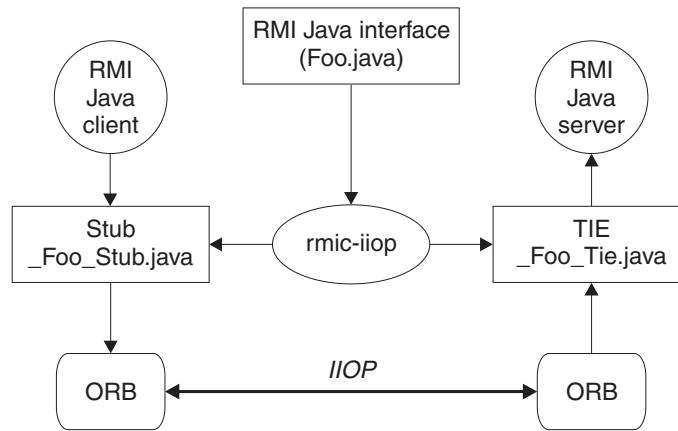


Figure 6. The ORB client side

The client relies on the existence of a server that contains an object that is that Foo interface. You must, therefore, create a proxy. This proxy is an object, called *stub* that acts as an interface between client application and ORB.

To create the stub, run the RMIC compiler on the Java interface: `rmic -iiop Foo`. This action generates a file/object that is named `_Foo_Stub`.

The presence of a stub is not always mandatory for a client application to operate. When you use particular CORBA features such as the DII (Dynamic Invocation Interface), you do not require a stub because 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 standalone Java application, the client has to create an instance of the ORB 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 that contains pairs property-value
- A Java Properties object

For an applet, a similar method is used in which a Java Applet is passed instead of the string array.

The first step of the ORB initialization is the processing of the ORB properties. The properties are processed in the following sequence:

1. 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 the `orb.properties` file that is in the `<user-home>` directory (if the file exists)

5. Check in the orb.properties file that is in the <java-home>/lib directory (if the file exists)
6. Fall back on a hardcoded default behavior

The two properties ORBClass and ORBSingletonClass determine which ORB class has to be instantiated.

The ORB loads its native libraries. Libraries are not mandatory, but they improve performance.

After this, the ORB starts and initializes the TCP transport layer. If the ListenerPort property was set, the ORB also opens a ServerSocket that is listening for incoming requests, as a server-side ORB usually does. At the end of the `init()` method, the ORB is fully functional and ready to support the client application.

### Getting hold of the remote object

Several methods exist by which the client can get a reference for the remote object. Usually, this reference is in a stringified form, called an IOR (Interoperable Object Reference). For example:

`IOR:0000000000000001d524d493a5.....`

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

Generally, the client ORB is not supposed to understand the details of the IOR, but use it as a sort of a key; that is, a reference to the remote object. However, when client and server are both using an IBM ORB, extra features are coded in the IOR. For example, the IBM ORB adds into the IOR a proprietary field that is called `IBM_PARTNER_VERSION`. This field looks like:

`49424d0a 00000008 00000000 1400 0005`

where:

- The three initial bytes (from left to right) are the ASCII code for IBM, followed by `0x0A`, which specifies that the following bytes handle the partner version.
- The next four bytes encode the length of the remaining data (in this case 8 bytes)
- The next four null bytes are for future use.
- The two bytes for the Partner Version Major field (`0x1400`) define the release of the ORB that is being used (1.4.0 in this case).
- The Minor field (`0x0005`) distinguishes in the same release, service refreshes that contain changes that have affected the backward compatibility.

Because the IOR is not visible to application-level ORB programmers and the client ORB does not know where to look for it, another step has to be made. This step is called the bootstrap process. Basically, the client application needs to tell the ORB where the remote object reference is located.

A typical example of bootstrapping is if you use a naming service: the client invokes the ORB method `resolve_initial_references("NameService")` that returns (after narrowing) a reference to the name server in the form of a `NamingContext` object. The ORB looks for a name server in the local machine at the port 2809 (as default). If no name server exists, or the name server is listening in another port, the ORB returns an exception. The client application can specify a different host, port, or both by using the `-ORBInitRef` and `-ORBInitPort` options.

## How the ORB works

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; that is, your \_Foo\_Stub.

ORB.resolve\_initial\_references() causes a lot of activity under the covers. Mainly, the ORB starts a remote communication with the name server. This communication might include several requests and replies. Usually the client ORB first checks whether a name server is listening, then asks for the specified remote reference. In an application where performance is considered important, caching the remote reference is a better alternative to repetitive use of the naming service. However, because the naming service implementation is a transient type, the validity of the cached reference is tied 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(). By invoking the previous two methods where the string parameter has a corbaloc (or corbaname) format as, for example:

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

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 Aservice that is requested, and returns 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

At this point, the client should hold a reference to the remote object that is an instance of the stub class. The next step is to invoke the method on that reference. The stub implements the Foo interface and therefore contains the message() method that the client has invoked. It is that method that is executed.

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

**Note:** In this discussion, the remote object will be called *FooImpl*, which in CORBA language is referred to as a *servant*.

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 invoking its \_request() method, specifying the name of the method to invoke and whether a reply is expected or not.

Note that the CORBA specification imposes an extra indirection layer between the ORB code and the stub. This layer is commonly known as *delegation*. CORBA imposes the layer by 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. This allows a stub that is generated by one vendor's ORB to work with the delegate from another vendor's ORB.

When creating a request, the ORB first checks whether the enableLocateRequest property is set to true. If it is, a LocateRequest is created. The steps of creating this request are similar to 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, object key) to create a connection if it does not already 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 at this point 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. Usually that involves the creation of a Java socket and a reader thread that is ready to intercept the server reply. The ORB.connect() is invoked as part of this process.

When the ORB has the connection, it proceeds to create the Request message. In the message are the header and the body of the request. The CORBA 2.3 specification specifies the exact format. The header contains, for example, local and remote IP addresses and ports, message size, version of the CORBA stream format (GIOP 1.x with x=0,1,2), byte sequence convention, request types, and Ids. (See Chapter 20, “ORB problem determination,” on page 169 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 call-back feature (see the CORBA specification), the second to specify the encoding of strings.

In the next step, the stub calls \_invoke(). Again it is the delegate invoke() method that is executed. The ORB in this chain of events calls the send() method on the connection that will write the request to the socket buffer and flush 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, demarshals it, and returns the correct object.

## The server side

Typically, a server is an application that makes available one of its implemented objects through an ORB instance. The subjects discussed here are:

- “Servant implementation” on page 54
- “Tie generation” on page 54
- “Servant binding” on page 54
- “Processing a request” on page 55

### 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 `Foo` interface. Here, the first case is described. From now, the servant is called `FooImpl`.

### Tie generation

Again, 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 invoking the `rmic` compiler. These classes are generated from the compiled Java programming language classes that contain remote object implementations; for example, `rmic -iiop FooImpl` generates the stub `_Foo_Stub.class` and the tie `_Foo_Tie.class`.

### Servant binding

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 FooImpl(...)`
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 invoking the `ORB` static method `init(...)`. The usual steps for that 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

Then, the server needs to create an instance of the servant class `FooImpl.class`. Something more than the creation of an instance of a class happens under the cover. Remember that the servant `FooImpl` extends the `PortableRemoteObject` class, so the constructor of `PortableRemoteObject` is executed. This constructor calls the static method `exportObject(...)` whose parameter is the same servant instance that you try to instantiate. The programmer must directly call `exportObject()` if it is decided that the servant will not inherit from `PortableRemoteObject`.

The `exportObject()` method first tries to load a rmi-iiop tie. The ORB implements a cache of classes of ties for improving performances. 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. It stops if it finds a `PortableRemoteObject` class or a `java.lang.Object`, and returns null. Otherwise, it returns an instance of that tie that is kept in a hashtable that is paired with the instance of the tie's servant. If the ORB cannot get hold of the tie, it guesses that an RMI (JRMP) skeleton might be present and calls the `exportObject()` method of the `UnicastRemoteObject` class. Finally, if all fails, a null tie and exception is thrown. At this point, the servant is ready to receive remote methods invocations. However, it is not yet reachable.

In the next step, the server code has to get hold of the tie itself (assuming the ORB has already done this successfully) 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, in turn, fetches 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 invokes the ORB method `resolve_initial_references("NameService")`. It then creates a NameComponent, a sort of directory tree object that identifies in the naming service the path and the name of the remote object reference, and binds together this NameComponent 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

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. As seen for the client side, the ORB caches all the connections that it creates.

By using the connection, the listener thread spawns 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), and delegates the reading of the message. The worker threads read all the fields in the message and dispatch them to the tie, which unmarshals any parameters and invokes 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. After that, the connection is removed from the reader thread which eventually stops.

## Features of the ORB

This section describes:

- “Portable object adapter”
- “Fragmentation” on page 57
- “Portable interceptors” on page 58
- “Interoperable naming service (INS)” on page 60

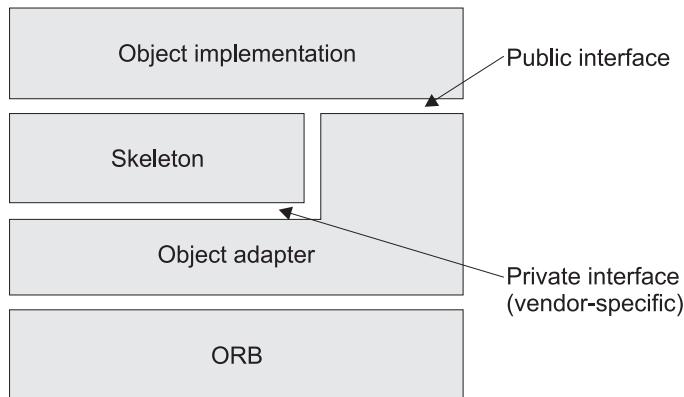
### Portable object adapter

An object adapter is the primary way for an object to access ORB services such as object reference generation. An object adapter exports a public interface to the object implementation, and a private interface to the skeleton. The main responsibilities of an object adapter are:

## ORB - features

- Generation and interpretation of object references
- Method invocation
- Object and implementation activation and deactivation
- Mapping object references to the corresponding object implementations

Figure 7 shows how the object adapter relates to the ORB, the skeleton, and the object implementation.



*Figure 7. Relationship between the ORB, the object adapter, the skeleton, and the object implementation*

In CORBA 2.1 and below, all ORB vendors had to implement an object adapter, which was known as the basic object adapter. Because the basic object adapter was never completely specified with a standard CORBA IDL, vendors implemented it in many different ways. Therefore, for example, programmers could not write server implementations that could be 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 for 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:

- Allow programmers to construct object and server implementations that are portable between different ORB products.
- Provide support for persistent objects; that is, objects whose lifetimes span multiple server lifetimes.
- Support transparent activation of objects and the ability to associate policy information to 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.

Since IBM J2SE v.1.4, the ORB supports both the POA specification and the proprietary basic object adapter that is already present in previous IBM ORB versions. As default, the rmic 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.

If you want to implement an object that is using the POA, the server application must obtain a POA object. When the server application invokes the ORB method

`resolve_initial_reference(RootPOA)`, the ORB returns the reference to the main POA object that contains default policies (see the CORBA specification for a complete list of all the POA policies). You can create new POAs as children of the RootPOA, and these children can contain different policies. This in turn allows you to manage different sets of objects separately, and to partition the name space of objects IDs.

Ultimately, a POA handles Object IDs and active servants. An active servant is a programming object that exists in memory and has been registered with the POA by use of one or more associated object identities. The ORB and POA cooperate to determine on which servant the client-requested operation should be invoked. 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 provides also a default servant that is used when no active servant has been registered. You can register a particular implementation of this default servant and also of a servant manager, which is an object for managing the association of an object ID with a particular servant. A simple POA architecture is represented in Figure 8.

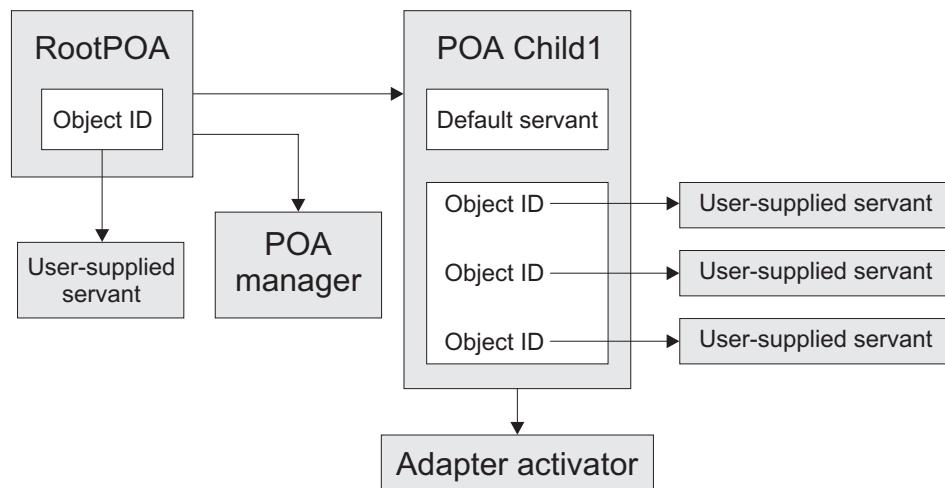


Figure 8. Simple portable object adapter architecture

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 marshaled 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 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

## ORB - features

trade-off in almost all conditions. The fragment time-out should not be set to too low a value, or time-outs might occur unnecessarily.

### Portable interceptors

CORBA implementations have long had proprietary mechanisms that allow users to insert their own code into the ORB's flow of execution. This code, known as interceptors, is called at particular stages during the processing of requests. It can directly inspect and even manipulate requests.

Because this message filtering mechanism is extremely flexible and powerful, the OMG standardized interceptors in the CORBA 2.4.2 specification under the name "portable interceptors". The idea is to define a standard interface to register and execute 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 and IOR interceptors. Request interceptors are called during request mediation. IOR interceptors are called when new object references are created so that service-specific data can be added to the newly-created IOR in the form of tagged components.

The ORB calls request interceptors on the client and the server side to manipulate service context information. Interceptors must register with the ORB for those interceptor points that are to be executed.

Five interception points are 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 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 interceptor point for IOR interceptors is `establish_component`. The ORB calls this interceptor 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. Registration of interceptors is done using the interface `ORBInitializer`.

Example:

```
package pi;

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 to ensure that an instance of this class does not get marshaled, because an interceptor instance is strongly tied to the ORB with which it is registered. This trivial implementation prints out a message at every interception point.

You can do a simple registration of the interceptor by using the **ORBInitializer** class. Because interceptors are intended to be a means by which ORB services access ORB processing, by the time the `init()` method call on the ORB class returns an ORB instance, the interceptors have already been registered. It follows that interceptors cannot be registered with an ORB instance that is returned from the `init()` method call.

First, you must create a class that implements the **ORBInitializer** class. This class will be called by the ORB during its initialization:

```
public class MyInterceptorORBInitializer extends LocalObject implements ORBInitializer {  
  
    public static Interceptor interceptor;  
    public String name() { return ""; }  
  
    public void pre_init(ORBInitInfo info) {
```

## ORB - features

```
try {
    interceptor = new MyInterceptor();
} 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 runtime obtains the ORB properties that begin with org.omg.PortableInterceptor.ORBInitializerClass;. The remaining portion 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)

CosNaming that is implemented in the IBM ORB is another name for the CORBA Naming Service that observes the OMG Interoperable Naming Service specification (INS, CORBA 2.3 specification). It 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 a new string format that can be passed as a parameter to the ORB methods `string_to_object()` and `resolve_initial_references()` such as the corbaname and corbaloc formats.

Corbaloc URIs allow you to specify object references that can be contacted by IIOP, or found through `ORB::resolve_initial_references()`. This new format is easier than IOR is to manipulate. To specify an IIOP object reference, use a URI of the form (see the CORBA 2.4.2 specification for full syntax):

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

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 (see the CORBA 2.4.2 specification) cause `string_to_object()` to look up a name in a CORBA naming service. They are an extension of the corbaloc syntax:

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

For example:

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

where the portion of the reference up to the hash mark (#) is the URL that returns the root naming context. The second part 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>. So, for example, with command-line arguments of:

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

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 specifically configured (with -ORBInitRef), it constructs a string that consists of the default prefix, a `/' character, and the name requested. The string is then fed to string\_to\_object(). So, 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.

You can specify **-ORBInitRef** and **-ORBDefaultInitRef** also as system properties; for example:

```
-Dcom.ibm.CORBA.ORBInitRef.NameService="corbaloc:..."  
-Dcom.ibm.CORBA.ORBDefaultInitRef="corbaloc:..."
```

## **ORB - features**

---

## Chapter 7. Understanding the Java Native Interface (JNI)

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 Inc maintains a combined programming guide and specification at <http://java.sun.com/docs/books/jni/>.

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

The topics that are discussed in this chapter are:

- “Overview of JNI”
- “The JNI and the Garbage Collector” on page 64
- “Copying and pinning” on page 65
- “Handling local references” on page 66
- “Handling global references” on page 69
- “Handling exceptions” on page 69
- “Using the `isCopy` flag” on page 69
- “Using the mode flag” on page 70
- “A generic way to use the `isCopy` and mode flags” on page 71
- “Synchronization” on page 71
- “Debugging the JNI” on page 72
- “JNI checklist” on page 73

---

### Overview of JNI

The JNI is a set of wrapper functions that enables C or C++ code to access Java code, and Java code to access C or C++ code. The JNI does very little management; it mostly provides a vehicle for the code.

**Note:** In this chapter, C/C++ code is always called native code because it runs directly on the target platform, unlike Java code, which requires a JVM.

You can use the JNI in two ways:

- You can write some C or C++ code in a library, and call it from your Java application.
- You can embed a JVM in your native application so that you can write some parts of that application in Java. This way is the normal runtime mode of Java; that is, you start a native Java executable, which then embeds a JVM to execute the Java code that you specify to that executable.

The JNI specification does not have a complete set of rules about how the JNI is to be implemented. Therefore, different vendors implement JNI in different ways. The Sun trademark specification and the Java Compatibility Kit (JCK) ensure compliance to the *specification*, but not to the *implementation*. Native code should conform to the specification and not to the implementation. Code written against the implementation is less portable than code written to the specification.

### The JNI and the Garbage Collector

Before you read about the two main JNI topics (“Handling local references” on page 66 and “Handling global references” on page 69), you need to understand why and how references are maintained, and how the Garbage Collector is involved.

Two main interactions occur between the Garbage Collector and the JNI:

1. Garbage Collector and object references
2. Garbage Collector and global references

These interactions manage Java objects in native code.

#### Garbage Collector and object references

The Garbage Collector reclaims garbage, which is defined as anything on the Java heap that is not reachable. However, if you access a Java object from your native code, the reference for that access might not exist in a form that the Garbage Collector can trace. The Garbage Collector, therefore, is likely to deduce that objects that you have referenced or created are garbage. The Garbage Collector can, from its root set of object pointers, trace only references to objects that are in the Java heap (see Chapter 2, “Understanding the Garbage Collector,” on page 7).

To avoid this problem, the JNI automatically creates a local reference to any object that is referenced across it. The local reference that it creates for your object is a pointer to your object. It is created in the stack of the thread that is running your code. When the Garbage Collector runs, it finds that local reference as part of its root set of object pointers (see Chapter 2, “Understanding the Garbage Collector,” on page 7) and therefore does not collect your object.

You can think of local references as invisible automatic variables that are in the function or method that you use to access a Java object. The invisible variable is passed on (invisibly) to all the functions that are called within the function that declares the local reference, and to all the functions that are called by them, and so on. As with all automatic variables, the local reference goes out of scope when you exit the function in which it was declared.

Therefore, you have two elements of data for objects to which you refer across the JNI. You have a *real object* that exists on the Java heap, and you have a *reference* to that object. This reference exists on the stack of your native thread. When the reference disappears, it does not directly affect the object to which you referred, but the object might become unreachable and therefore able to be collected by a future garbage collection cycle. An object can have more than one native reference to it, and remains uncollectable as long as one or more references exist.

Here is some JNI code:

```
static void JNICALL (...)
{
    jobject myObject = env->NewObject ()
    env->GetObjectClass (myObject)
}
```

Here is how the same code would look if you used a local variable to create an object reference (invisible code is in *italics*):

```
static void JNICALL (...)  
{  
    void * myObjectLocalRef;  
  
    jobject myObject = env->NewObject ()  
    myObjectLocalRef = *myObject  
  
    env->GetObjectClass (myObject, myObjectLocalRef)  
  
}// myObjectLocalRef goes out of scope here
```

The `myObjectLocalRef` is created in the scope of the function or method that creates the object for which the local reference exists. This imaginary automatic variable refers to `myObject` so that it cannot be garbage collected in the scope of the local reference. The analogy has been expanded a little by the passing of the automatic variable into all the functions that are called inside the scope. The idea is that the local reference in `JNICALL` remains active in the `GetObjectClass` function, and in any other functions that it calls. Only when you exit the function (or method) in which a local reference is created does it become invalid (or out of scope). How this affects your application is discussed in more detail in “Handling local references” on page 66.

## Garbage Collector and global references

“Garbage Collector and object references” on page 64 showed how local references are automatically created and deleted. The scope of local references, however, is limited. If you want to use an object outside the scope of a local reference, you must manually create a reference to it. Obviously, you are also responsible for deleting such a reference. These references are known as global references. Global references are stored in a space that is reserved by the JVM. This space is in the native heap space for the Java process. The Garbage Collector always checks in this special space to determine whether a reference exists to an otherwise unreachable object.

Another class of references is available. These references are known as weak global references whose typical function is to cache objects. For more information about weak global references, see your JNI documentation.

---

## Copying and pinning

Objects that are on the Java heap are usually mobile; that is, the Garbage Collector can move them around if it decides to resequence the heap. Some objects, however, cannot be moved either permanently, or temporarily. Such immovable objects are known as *pinned* objects.

When native code, by way of the JNI, creates or refers to an object that is on the heap, a JVM can do either of these actions:

- Make a copy of the object in local storage, and return this copy to the caller
- Pin the actual object on the heap, and return a pointer to the caller

The action taken depends on the Java Virtual Machine's JNI implementation. The IBM Virtual Machine for 5.0 uses a copying implementation.

The caller is told whether the object is a copy or is pinned, by way of a flag in the appropriate API call.

## Handling local references

You must understand the scoping rules of local references before you can understand the problems that this section discusses. Ensure that you have read “The JNI and the Garbage Collector” on page 64 or have visited the Sun Web site at <http://www.sun.com> and read the documentation or specification that is given there.

### Local reference scope

It is very easy to lose a local reference accidentally. That is, the local reference goes out of scope, but you continue to use the objects to which it used to refer. When you lose a local reference in this way, problems will occur later. The loss of a local reference does not invalidate the object to which it refers. Your application continues to work normally and to use the object, until a garbage collection cycle occurs. However, until the space on the heap is moved or reused, you can continue to use the object. Your code is pointing to invalid space, but that space continues to hold the valid data that you put into it.

So your application might seem to work well, but, at random intervals, it fails when an object that you think is valid suddenly disappears. This is the type of problem that usually occurs late in a product cycle. It can be quite difficult to isolate. If you always have this type of problem shortly after a garbage collection cycle with compaction, when objects are moved, it is a good hint that local references are being misused.

Consider this example code:

```
jobject myJNIfunc1 ()  
{  
    return env->NewObject ()  
}  
  
void myJNIfunc2 ()  
{  
    jobject obj;  
  
    obj = myJNIfunc1 ()  
  
    ..  
    ..  
}
```

When an object is created in `myJNIfunc1`, a local reference is created. This reference immediately goes out of scope when `JNIfunc1` returns. When `obj` is set in `myJNIfunc2`, no local reference to `obj` exists, and the Garbage Collector can collect it.

The reason for this is that the references to objects in the C code are not references that the Garbage Collector normally follows. They are C or C++ type references that are generated by the native compiler and are, therefore, not truly pointers into the Java heap, which are what the Garbage Collector uses to find its root set of objects. The native reference to a Java object is translated appropriately in a JNI function. So, to make it work, the JVM creates a special stack frame when a JNI function is entered, and reserves a set number of locations in the frame for any pointers (local references) that the function might need. When the Garbage Collector looks for root objects, it always looks at this area of the stack frame.

This is how it works:

- When the call to `NewObject` returns, the JNI function `NewObject` has created a local reference for you and stored it in the stack frame. Now, the stack has a pointer to the object that the Garbage Collector can see (see Figure 9).

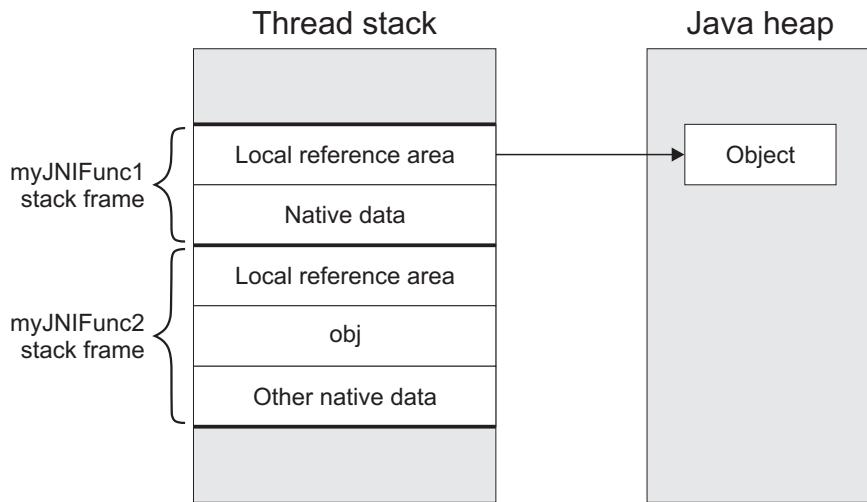


Figure 9. Thread stack pointing to an object so that the Garbage Collector can see the object

- In Figure 10, function `myJNIFunc1` has returned, so the `myJNIFunc1` stack frame has been popped, and the local reference is therefore lost. A reference to `Object`, to which `obj` can refer, exists on the stack (), and JNI functions can use this reference to reach `object` on the Java heap. However, the stack contains no reference that the Garbage Collector can see, that points to the Java heap object. Therefore, the Java heap object is unreachable and eligible for garbage collection.

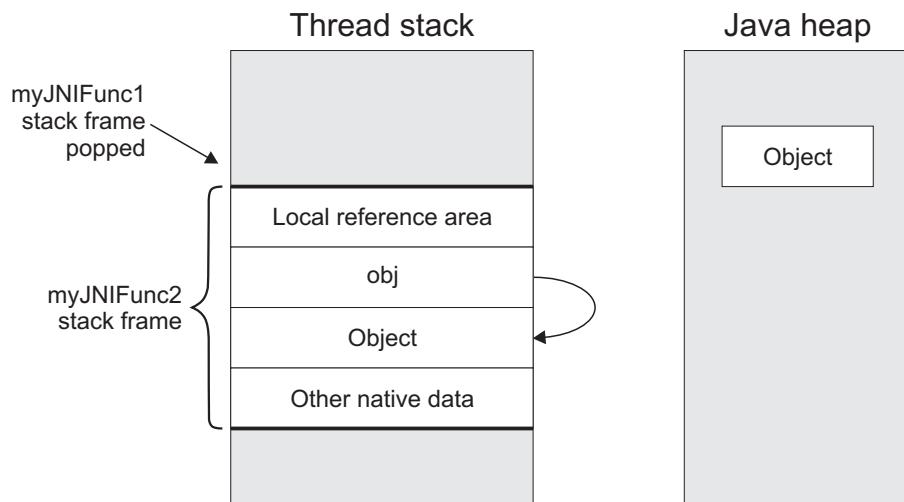


Figure 10. Thread stack not pointing to an object so that the Garbage Collector cannot see the object

The use of `obj` now can be fatal because a garbage collection cycle can overwrite the data to which `obj` refers. Clearly, a local reference operates like an automatic reference. You cannot rely on it outside the function in which the local reference was generated.

## Local reference scope

Another incorrect way to refer to an object outside the scope of its local reference is:

```
static jclass cls = 0;

void myJNIFunc
{
    ...
    if (cls == 0)
    {
        cls = (*env)->GetObjectClass(env, obj);
        if (cls == 0)
        {
            ... /* error */
        }
    }
    // there's no local ref to cls at this point
    ...
}
```

The error occurs because any local reference to `cls` goes out of scope on the first exit from `myJNIFunc`. Therefore, the object could be garbage collected although the static variable, which the Garbage Collector cannot examine, still contains the reference. Again, no area in the stack directly refers to `cls`.

## Summary of local references

Local references cannot be shared between separate functions or methods. Because local references are like automatic variables, you cannot share them between threads.

## Local reference capacity

The JNI specification dictates that each native method can safely create up to 16 local references. If a method requires more than 16, the routines should use `PushLocalFrame` or `EnsureLocalCapacity` to ensure that the references can be created safely and also ensure portability across virtual machine implementations.

The IBM virtual machine silently allows more than 16 references to be created unless the `-Xcheck:jni` option is used. If `-Xcheck:jni` is specified, a warning is issued if more than 16 local references are used. The warning identifies the native method that triggered the overflow. The warning looks something like this:

```
JVMJNCK065W JNI warning in NewLocalRef: Automatically grew local ref frame capacity from 16 to 1013. 17 refs are in use. Use EnsureLocalCapacity or PushLocalFrame to explicitly grow the frame.
```

```
JVMJNCK078W Warning detected in com/acme/MyClass.myNative(III)V
```

## Manually handling local references

You can control the storage capacity and freeing of local references, but you cannot control whether they are created or not. You can create extra local references if you want to. IBM strongly recommends that you do not create new local references in an attempt to keep an object alive outside its automatic local reference scope. If you do, it is almost certain that a window will remain through which data is lost in a garbage collection cycle. Use global references instead.

Ensure that you do not refer to an object after you delete its local reference unless you have a global reference to it. It might be good housekeeping to throw away a local reference to an object when you have attached a global reference to it.

## Handling global references

Use a global reference to refer to a JNI object where the scope of the local reference is too restricted. You can use global references across threads and between functions and methods. The Garbage Collector always finds objects that are accessed through global references. Every “create global reference” call must have a corresponding “free global reference” call. Otherwise, the global references accumulate and cause a memory leak, because the objects that they reference are never collected. The JVM does not (cannot) police or check global references. Global references are completely under the JNI programmer’s control.

Leaks in global references eventually lead to an out-of-memory exception. They can be quite difficult to solve, especially if you do not manage JNI exception handling (see “Handling exceptions”).

## Global reference capacity

The JNI specification does not define what the capacity of the JVM to hold global references should be. The IBM Virtual Machine for Java has a fairly small limit, on the order of  $10^3$ . Other JVMs have a much larger capacity or perhaps an unlimited capacity (subject only to overriding process or platform sizes). This implementation detail can cause problems. If you have a reference leak, it might not show up for a very long time on some JVMs, although it will eventually. That same leak would show up much more quickly on the IBM Virtual Machine for Java. This difference can lead you to think mistakenly that your application works on the vendor’s JVM, but not on the IBM Virtual Machine for Java.

## 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 because it 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. Note that a JNI function with a return code always sets an error if an exception is pending. That is, you do not need to check for exceptions if a JNI function returns “success”, but you do need to check for an exception in an error case. If you do not check, the next time you go through the JNI, the JNI code will detect a pending exception and throw it. Clearly, 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 actually created.

**Note:** The JNI `ExceptionCheck` function might be a cheaper way of doing exception checks than the `ExceptionOccurred` call, because the `ExceptionOccurred` call has to create both an object to which you can refer, and a local reference.

## Using the `isCopy` flag

Many of the JNI functions have a copy flag as a parameter (`jboolean *isCopy`). On return, the flag is set to state TRUE if the data that is returned is a copy, or to FALSE if that data is pinned. Whether to copy or pin data is an implementation detail (see “Copying and pinning” on page 65).

## JNI - using the isCopy flag

The IBM Virtual Machine for Java uses the copy implementation.

The isCopy flag is an output parameter. You cannot set it, on entry to a JNI function, to specify whether you want copy or pin. You do not have to use this flag at all. You can pass NULL into the JNI function to indicate that you do not care what the result is.

If the flag indicates a copy, a copy of the data has been taken. If the flag indicates pinning, the data that is on the heap has been marked as referenced and pinned. Pinned data cannot be moved in a compaction cycle, nor collected. If the data is pinned, you effectively have a direct pointer to the data that is on the Java heap.

Clearly, you must free the space that is used for a copy of the data. Also, you must free the data when it is pinned. By doing this, you tell the JVM that it can unpin the data again. For example, the GetBooleanArrayElements call must always be followed by a ReleaseBooleanArrayElements call, whatever the setting of the isCopy flag.

A common mistake is to think that only copied data needs to be freed. If you assume that you need free only data that is copied, the heap gradually becomes more and more fragmented with bits of uncollectable, pinned data. Eventually, a failure occurs.

Use of the isCopy flag is one of the JNI specification details in which you might accidentally code to a JVM that prefers the copy method. Everything works correctly if you accidentally free only copied data. If you swap to a pinning JVM (or the JVM that you use changes its algorithm), code that was working fails if it is not written to specification.

The JNI specification also states: “It is not possible to predict whether any given JVM will copy or pin data on any particular JNI call”. If the flag indicates that a copy has been used, another trap opens in which you must be sensitive to the mode flag in the corresponding release call (see “Using the mode flag”).

Always call the Release<something> function after a function that is using the isCopy flag.

---

## Using the mode flag

This flag is used in Release<something>Array calls. For example:

```
ReleaseBooleanArrayElements  
(JNIEnv *env, jbooleanArray array, jboolean *elems, jint mode);
```

You must use this flag correctly with respect to the setting of the corresponding isCopy flag. You need to know what the isCopy flag is telling you (see “Using the isCopy flag” on page 69). If the isCopy flag indicates that the returned data is pinned, any preceding changes that you made to the data have been copied directly into the Java heap, and the mode parameter is ignored.

If, however, the isCopy flag indicates that the returned data is a copy, you must use the mode flag to ensure that all changes that you made are actually actioned.

The possible settings of the mode flag are:

- 0 Update the data on the Java heap, do not free the copy.

**JNI\_COMMIT**

Update the data on the Java heap and free the space used by the copy.

**JNI\_ABORT**

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

If you do not change the array data that you got as a copy, use JNI\_ABORT because it prevents unnecessary copying. If you do change the data, use 0 or JNI\_COMMIT to ensure that your changes actually happen, or use JNI\_ABORT if appropriate.

- If the isCopy flag indicates that the data is pinned, use the JNI\_ABORT setting.
- If the isCopy flag indicates that the data is a copy, use the appropriate setting.

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

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

- 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 if you want to do some special optimization. This generic way does *not* release you from the need to think about synchronization (see “Synchronization”).

## Synchronization

When you get array elements through a Get<something>ArrayElements call, you must think about synchronization. Whether or not the data is pinned, 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

It is likely that these two entities are 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, note that the code *would* work with a pinning JVM. Because each thread writes only its own bit of the data and the mode flag is

## JNI - synchronization

ignored, no contention occurs. This is another example of how code that is not strictly to specification would work with one JVM implementation and not with another. Although this scenario involves an array elements copy, you can see that pinned data can also be corrupted when two threads access it at the same time. Take care if the getter method says the data is pinned.

Be very careful about how you synchronize access to array elements. The JNI interfaces allow you to access regions of Java entities to reduce problems in this sort of interaction. In the above scenario, the thread that is writing the data should write into its own region, and the thread that is reading the data should read only its own region. This works whatever the JNI implementation is.

---

## Debugging the JNI

Errors in JNI code can occur in several ways:

- The program crashes while it is executing 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 diagnostics that help you check the JNI transitions. To invoke these diagnostics, specify the **-Xcheck:jni** option when you start up 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.
- The type matching, in get or set field operations.
- The validity of static and nonstatic field IDs.
- Whether strings are valid and non-null.
- Whether array elements are non-null.
- The types on array elements.

Output from jnichk appears on the standard output stream, and looks like:

```
JNI warning in FindClass: argument #2 is a malformed identifier ("invalid.name")
Warning occurred 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:<sub-option>[,<...>]**. Useful suboptions are:

<b>all</b>	Check application and system classes
<b>verbose</b>	Trace certain JNI functions and activities
<b>trace</b>	Trace all JNI functions
<b>nobounds</b>	Do not perform bounds checking on strings and arrays
<b>nonfatal</b>	Do not exit when errors are detected
<b>nowarn</b>	Do not display warnings
<b>noadvice</b>	Do not display advice
<b>novalist</b>	Do not check for va_list reuse (see note below)
<b>pedantic</b>	Perform more thorough, but slower checks
<b>help</b>	Print help information

The **-Xcheck:jni** option introduces some overhead because it is very thorough when it validates input parameters.

**Note:** On some platforms, reusing a va\_list in a second JNI call (for example, when calling `CallStaticVoidMethodV()` 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`.

## JNI checklist

Table 4. JNI checklist

Remember	Outcome of nonadherence
Check your code to ensure that you do not accidentally lose local references. If in doubt, create a global reference and ensure that you delete that global reference when appropriate.	Random crashes (depending on what you pick up in the overwritten object space) happen at random intervals.
Local references cannot be saved in global variables.	As above.
Do not attempt to manipulate local references.	As above. This problem might occur only in small windows, very infrequently.
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.

## JNI checklist

*Table 4. JNI checklist (continued)*

Remember	Outcome of nonadherence
Always check for exceptions (or return codes) on return from a JNI function. Always handle a deferred exception immediately you detect it.	Unexplained exception in apparently perfect code
Ensure that array and char 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 71).	Memory leaks, heap fragmentation, or both.
When you update a Java object in native code, ensure synchronization of access.	Memory corruption.

---

## Chapter 8. Understanding 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, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism.

---

### The RMI implementation

The RMI implementation consists of three abstraction layers:

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 below 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.

## The RMI implementation

- 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

As explained in the previous section, on the server side, 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.TCPTTransport` class. Thread pooling is not enabled by default. Enable it with this command-line setting:

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

(or 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. Because you cannot currently fine tune the number of threads in the pool, 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 and there is no overhead if these threads are created for every RMI call.

---

## Understanding Distributed Garbage Collection (DGC)

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

The DGC abstraction is used for the server side of Distributed Garbage Collection. This interface contains two methods: `dirty()` and `clean()`. A `dirty()` call is made when a remote reference is unmarshalled in a client (the client is indicated by its VMID). A corresponding `clean()` call is made when no more references to the remote reference exist in the client. A failed `dirty()` call must schedule a strong `clean()` call so that the call's sequence number can be retained in order to detect future calls received out of order by the distributed garbage collector.

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

`DGCCClient` implements the client side of the RMI Distributed Garbage Collection system. The external interface to `DGCCClient` is the `registerRefs()` method. When a `LiveRef` to a remote object enters the JVM, it must be registered with the `DGCCClient` to participate in distributed garbage collection. When the first `LiveRef` to a particular remote object is registered, a `dirty` call is made to the server-side distributed garbage collector for the remote object, which returns a lease

guaranteeing that the server-side DGC will not collect the remote object for a certain period of time. While LiveRef instances to remote objects on a particular server exist, the DGCCClient periodically sends more dirty calls to renew its lease. The DGCCClient tracks the local availability of registered LiveRef instances using phantom references. When the LiveRef instance for a particular remote object is garbage collected locally, a `clean()` call is made to the server-side distributed garbage collector, indicating that the server no longer needs 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 would wait until the next lease renewal or until any phantom reference is queued for generating clean requests as necessary.

---

## Debugging applications involving RMI

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 at the network level.



---

## **Part 2. Submitting problem reports**

This part describes how to gather data about a problem and how to send that data to IBM service.

The chapters are:

- Chapter 9, “Overview of problem submission,” on page 81
- Chapter 10, “MustGather: Collecting the correct data to solve problems,” on page 83
- Chapter 11, “Advice about problem submission,” on page 87
- Chapter 12, “Submitting data with a problem report,” on page 89



---

## Chapter 9. Overview of problem submission

This chapter gives an overview of Java service and how you can send problem reports.

---

### How does IBM service Java?

Java is not a product that IBM sells; it is a supporting technology.

No traditional level 1, level 2, and level 3 service exists for Java. However, the Java Technology Centre (JTC) maintains a Java L3 service team. Initially, your problem report will probably go to the L2 service team for the product that you are using. They will forward to the JTC if necessary. You can also send problem reports direct to the JTC, as described in this part of the book.

Java L3 service is in Hursley (England), Bangalore (India), and Ottawa (Canada). This geographical split is transparent to you for the purpose of submitting problem reports. However, if you have to communicate directly with a service engineer, be aware that:

- Hursley operates on GMT and uses Daylight Savings Time (DST).
- Bangalore operates on Indian Standard Time (IST), which is GMT + 4.5 and does not use DST.
- Ottawa operates on Eastern Standard Time (EST), which is GMT -5 and uses DST.

---

### Submitting Java problem reports to IBM

Three methods are available:

- **Create a Problem Management Report (PMR):** If you are inside IBM, you can do this directly. Your PMR will arrive on the Java PMR queue. If you are outside IBM, your IBM representative will do this for you. As noted above, a PMR might be created against the product that you are using. The product service team will forward that PMR to the JTC if L3 Java analysis is required. If you are outside IBM and would like access to the PMR system, ask your IBM representative for details.
- **By the web:** This route is available only if you have access to the IBM intranet. Go to <http://eureka.hursley.ibm.com>. This is a front end to the PMR system. Fill in the form, and the server will create a PMR for you and queue it directly to the Java queue.
- **Direct contact:** If you have direct contacts in the JTC, you can use them. However, this is not the most desirable route because you are dependent on one engineer, and that engineer might be absent for various reasons.

---

### Java duty manager

A Java duty manager is available 24 hours per day, seven days per week. The duty manager will call out staff if necessary. To call out the duty manager, you must have a PMR number. Ask your IBM representative for the telephone number of the Java duty manager.



---

## Chapter 10. MustGather: Collecting the correct data to solve problems

This chapter gives general guidance about how to generate a problem report and which data to include in it:

- “Before you submit a problem report”
- “Data to include”
- “Things to try” on page 84
- “Factors that affect JVM performance” on page 84
- “Test cases” on page 84
- “Performance problems – questions to ask” on page 85

See Part 3, “Problem determination,” on page 93 for specific information for your platform.

---

### Before you submit a problem report

To obtain a quicker response to your problems, you must try all the suitable diagnostics and provide as much information as possible. By doing this, you ensure that your initial submission contains the maximum information for IBM support to track down your problem. If all the data is not there, you will get a request for more information from IBM support and, therefore, increase the turnaround time.

---

### Data to include

The following checklist describes the information that you could include in your problem report:

#### Environment information

Always submit the following information:

- Version information (**-version** from the command line).
- Command-line options.
- Environment, non-default settings.
- OS and OS version.
- OS distribution (if applicable).

#### Dump files

Submit the following diagnostic files if they were generated:

- `javacore.<date>.<time>.<pid>.txt`
- `heapdump.<date>.<time>.<pid>.phd`
- `core.<date>.<time>.<pid>.dmp`
- `Snap<seq>.<date>.<time>.<pid>.trc`
- CEDUMP and transactional dumps for z/OS

The name formats of these files can vary from the formats shown depending on the use of **-Xdump** options and environment variables.

## Data to include

If a core.<date>.<time>.<pid>.dmp file is available, run jextract (described in Chapter 28, “Using the dump formatter,” on page 225) against the file to generate a core.<date>.<time>.<pid>.dmp.zip file. Submit that zip file.

On z/OS, you must copy the transactional dump to HFS before you can run jextract against it.

If a Snap<seq>.<date>.<time>.<pid>.trc file is available, run java com.ibm.jvm.format.TraceFormat (described in “Using the trace formatter” on page 298) against the file to generate a Snap<seq>.<date>.<time>.<pid>.trc.fmt file. Submit both files.

### Other diagnostics

- Verbose outputs
- Data from any diagnostics that you run
- Data from JIT diagnostics
- Platform-specific data

For information on how to gather this data, see Part 3, “Problem determination,” on page 93.

---

## Things to try

Refer to Chapter 13, “First steps in problem determination,” on page 95.

---

## Factors that affect JVM performance

- Heap and stack sizes (-Xms, -Xmx , -Xss, and -Xoss settings). The values that are being used can be obtained by the **-verbose:sizes** option.
- Memory settings (such as the large page usage with **-Xlp** and **MAXDATA** settings for AIX).
- The search path to the class libraries (class path, mostly used classpath should come first).
- Garbage collection activity.
- The quality of the application code.
- Just-in-Time compiler.
- The machine configuration:
  - I/O disk size and speed
  - Number and speed of CPUs
  - Processor cache size and speed
  - Random access memory size and speed
  - Network and network adapters number and speed

---

## Test cases

It is easier for IBM Service to solve a problem when a test case is available. Include a test case with your problem report wherever possible.

If your application is too large or too complex to reduce into a test case, provide, if possible, some sort of remote login so that IBM can see the problem in your environment. (For example, install a VNC/Remote Desktop server and provide logon details in the problem report.) This option is not very effective because IBM has no control over the target JVM.

If no test case is available, analysis takes longer. IBM might send you specially-instrumented JVMs that require the collection of the diagnostics data while you are using them. This method often results in a series of interim fixes, each providing progressively more instrumentation in the fault area. This operation obviously increases the turnaround time of the problem. It might be quicker for you to invest time and effort into a test case instead of having a costly cycle of installing repeated JVM instrumentation onto your application.

## Performance problems – questions to ask

When someone reports a performance problem, it is not enough only to gather data and analyze it. Without knowing the characteristics of the performance problem, you might waste time analyzing data that might not be related to the problem that is being reported.

Always obtain and give as much detail as possible before you attempt to collect or analyze data. Ask the following questions about the performance problem:

- Can the problem be demonstrated by running a specific test case or a sequence of events?
- Is the slow performance intermittent?
- Does it become slow, then disappear for a while?
- Does it occur at particular times of the day or in relation to some specific activity?
- Are all, or only some, operations slow?
- Which operation is slow? For example, elapsed time to complete a transaction, or time to paint the screen?
- When did the problem start occurring?
- Has the condition existed from the time the system was first installed or went into production?
- Did anything change on the system before the problem occurred (such as adding more users or upgrading the software installed on the system)?
- If you have a client and server operation, can the problem be demonstrated when run only locally on the server (network versus server problem)?
- Which vendor applications are running on the system, and are those applications included in the performance problem? For example, the IBM WebSphere Application Server?
- What effect does the performance problem have on the users?
- Which part of your analysis made you decide that the problem is caused by a defect in the SDK?
- What hardware are you using? Which models; how many CPUs; what are the memory sizes on the affected systems; what is the software configuration in which the problem is occurring?
- Does the problem affect only a single system, or does it affect multiple systems?
- What are the characteristics of the Java application that has the problem?
- Which performance objectives are not being met?
- Did the objectives come from measurements on another system? If so, what was the configuration of that system?

Two more ways in which you can help to get the problem solved more quickly are:

- Provide a clear written statement of a simple specific example of the problem, but be sure to separate the symptoms and facts from the theories, ideas, and

## Performance problems – questions

your own conclusions. PMRs that report “the system is slow” require extensive investigation to determine what you mean by slow, how it is measured, and what is acceptable performance.

- Provide information about everything that has changed on the system in the weeks before the problem first occurred. By missing something that changed, you can block a possible investigation path and delay the solution of the problem. If all the facts are available, the team can quickly reject those that are not related.

---

## Chapter 11. Advice about problem submission

This chapter describes how to submit a problem report, and explains the information that you should include in that report:

- “Raising a problem report”
- “What goes into a problem report?”
- “Problem severity ratings”
- “Escalating problem severity” on page 88

---

### Raising a problem report

See “Submitting Java problem reports to IBM” on page 81.

---

### What goes into a problem report?

- All the data that you can collect; see below
- Contact numbers
- A brief description of your application and how Java is part of it
- An assessment of the severity of the problem

---

### Problem severity ratings

Here is a guide to how to assess the severity of your problem. You can attach a severity of 1, 2, 3, or 4 to your problem, where:

#### Sev 1

- **In development:** You cannot continue development.
- **In service:** Customers cannot use your product.

#### Sev 2

- **In development:** Major delays exist in your development.
- **In service:** Users cannot access a major function of your product.

#### Sev 3

- **In development:** Major delays exist in your development, but you have temporary workarounds, or can continue to work on other parts of your project.
- **In service:** Users cannot access minor functions of your product.

#### Sev 4

- **In development:** Minor delays and irritations exist, but good workarounds are available.
- **In service:** Minor functions are affected or unavailable, but good workarounds are available.

An artificial increase of the severity of your problem does not result in quicker fixes. IBM queries your assessed severity if it seems too high. Problems that are assessed at Sev 1 require maximum effort from the IBM Service team and, therefore, 24-hour customer contact to enable Service Engineers to get more information.

---

## **Escalating problem severity**

For problems below Sev 1, ask IBM Service to raise the severity if conditions change. Do this, for example, when you discover that the problem is more wide-ranging than you first thought, or if you are approaching a deadline and no fix is forthcoming, or if you have waited too long for a fix.

For problems at Sev 1, you can escalate the severity higher into a 'critsit'. This route is available only to customers who have service contracts and to internal customers.

---

## Chapter 12. Submitting data with a problem report

After you have followed the advice that is given in the previous two chapters, you probably have a large amount of data to send to IBM in one or more files. In general, you will have received instructions within the PMR detailing the data to be collected and the mechanism for submitting it to IBM.

This chapter describes the two standard ways of providing that data to help problem determination by the IBM Java service team:

- IBM maintains an anonymous ftp public server, named 'ftp.emea.ibm.com', for sending or receiving data.
- You can also use an ftp server of your own if you want to. In your PMR, include details of how to log on, and where the data is. Java service might need to send data to you; for example, an interim fix (see "When you will receive your fix" on page 91). IBM uses the same server to send (PUT) data as Java service did to receive (GET) it. If you use your own server, provide an address that Java service can use to write to your server.

This chapter includes:

- "Sending files to IBM support"
- "Getting files from IBM support" on page 90
- "Using your own ftp server" on page 91
- "Compressing files" on page 91
- "When you will receive your fix" on page 91

---

### Sending files to IBM support

The preferred mechanism for sending files to IBM support is the IBM EMEA Centralized Customer Data Repository. For full information about this repository, see <http://www.ibm.com/de/support/ecurep/index.html>. If this mechanism cannot satisfy your requirements, discuss alternative arrangements with Java service. The basic mechanism involves:

1. ftp to ftp.emea.ibm.com (using a userid of "anonymous").
2. Change to the directory relevant to your platform. For example:
  - For AIX, change to toibm/aix
  - For Linux, change to toibm/linux
  - For Windows, change to toibm/windows
  - For z/OS, change to toibm/mvs
3. Set binary mode.
4. PUT your file using the standard naming convention detailed in the sample session below.

Your ftp session should resemble the one shown below:

```
ftp ftp.emea.ibm.com
Connected to mcevs1.mainz.de.ibm.com.
220-FTPSERVE IBM FTP CS V1R4 at MCEVS1.MAINZ.DE.IBM.COM, 11:49:14 on 2005-10-03.
220-Welcome to the IBM Centralized Customer Data Repository (ECuRep)
220-By using this service, you agree to all terms of the
220-Service User Licence Agreement
220-(see http://www.ibm.com/de/support/ecurep/service.html)!
```

## submitting data with a problem report

```
220-For FAQ/Documentation please see
220-http://itcenter.mainz.de.ibm.com and
220-http://www.ibm.com/de/support/ecurep/index.html
220- LOGIN user: anonymous pw: your_email_address
220-please report questions to: ftp.emea@mainz.ibm.com
220 Connection will close if idle for more than 15 minutes.
User (mcevsl.mainz.de.ibm.com:(none)): anonymous
331 Send email address as password please.
Password:
230-Here you can deliver/get support material to/from IBM.
230-Directories for:
230- deliver use command 'cd toibm'
230- get use command 'cd fromibm'
230- for CADCAM/CATIA/VPM/ENOVIA/SMARTTEAM use command 'cd cadcam'
230-
230-Please use command 'bin' prior transfer. See special instructions
230-displayed when changing to the sub directory.
230 'ANONYMOUS' logged on. Working directory is "/".
ftp> cd toibm
250-Here you can deliver Support Material to IBM.
250-Directories for: aix, cadcam, hw, linux, mvs, os2,
250-os400, swm, unix, vm, vse, windows.
250-
250-To enter the folder of your operating-system type 'cd'
250-Example: To enter the folder AIX type 'cd aix'.
250-Please use command 'bin' prior transfer.
250=====
250- IMPORTANT : only use the following characters for filenames:
250- Upper- or lowercas (A-Z), numbers (0-9),
250- period (.) and hyphen (-)
250- ==> Using other characters may lead to UNPREDICTABLE RESULTS,
250- ==> your file may NOT be processed |
250- E.g. Do NOT use BLANK characters, $-sign etc. in FILE NAMES |
250=====
250 HFS directory /toibm is the current working directory.
ftp> cd windows
250-Here you can place WINDOWS related support material for IBM
250-
250-Please follow our filenaming conventions in order to process your files
250-xxxxx.bbb.ccc.yyy.yyy ---> xxxx = PMR-Number
250- bbb = Branch Office (if known)
250- ccc = IBM Country Code (f.e. Germany 724)
250- yyy.yyy = Short description for the file type
250- f.e. tar.Z, restore.Z, restore.gz
250-Take care to use the binary Option before transfer.
250-Some additional Remarks:
250-1.) If possible inform your IBM Software Support about the files
250- transferred. This will reduce the reaction Time.
250-2.) The Material will be automatically deleted after 3 Working days.
250-3.) The FTP GET und LS option are intentionally disabled.
250 HFS directory /toibm/windows is the current working directory.
ftp> bin
200 Representation type is Image
ftp> put 12345.123.456.temp.zip
200 Port request OK.
125 Storing data set /toibm/windows/12345.123.456.temp.zip
250 Transfer completed successfully.
ftp: 11 bytes sent in 0.00Seconds 11000.00Kbytes/sec.
```

---

## Getting files from IBM support

To solve your problem temporarily or to provide extra materials to help diagnose the problem, IBM might need to send data to you. This data is sent using the reverse of the mechanism described above, by ftp-ing to the same server, going to the relevant fromibm directory and using GET to download the appropriate files to

your machine. Your PMR will normally provide the instructions for this operation. Remember that the files are available on the server for only a short time.

---

## Using your own ftp server

1. Dump the files and include the server address and log-in data in your problem report.
2. Give read and write access to IBM service for this area of your server.

---

## Compressing files

Core files (dumps) and trace files are often exceedingly large and you should compress them before sending them to IBM service using the appropriate platform tools. These tools support packaging of multiple files in one compressed file and you should take advantage of this packaging when appropriate to avoid multiple transfers and possible confusion. On Windows, use winzip or its equivalent. On UNIX platforms, use tar or gzip.

The output from the jextract mechanism (see Chapter 28, “Using the dump formatter,” on page 225) is usually a compressed file and this file (if sent by itself) does not need further compression.

**Do not send the files to IBM as attachments to e-mails.**

---

## When you will receive your fix

Java builds are performed daily at IBM. When an engineer has identified your problem and produced a fix, that fix goes into the overnight build.

IBM periodically produces service refreshes of Java. After you have been notified that your problem has been solved, you must obtain the next service refresh.

Service refreshes are fully supported by IBM. The version number in your JVM (see Part 3, “Problem determination,” on page 93) identifies the service refresh level that you are using. In some cases (for example when you urgently need a fix for a Sev 1 problem), IBM service provides you with an overnight build as an electronic fix (interim fix). An interim fix is a set of the Java binaries that contains a fix for your problem. IBM support sends you this set of binaries to replace your original binaries. Interim fixes are ftp’d to you through the same server that you used to send in your problem data. Interim fixes are used to validate that a fix is good in your environment, or to allow you to continue work on your project while waiting for the next service refresh. Interim fixes are *not* supported by Java service, because they have not been officially certified as Java-compatible. If you receive an interim fix, you must get the next service refresh immediately it becomes available.



---

## Part 3. Problem determination

This part of the book is the problem determination guide. It is intended to help you find the kind of fault you have and from there to 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

To use this part, go to the chapter that relates to your platform. If your application runs on more than one platform and is exhibiting the same problem on them all, go to the chapter about the platform to which you have the easiest access.

- Chapter 20, “ORB problem determination,” on page 169
- Chapter 21, “NLS problem determination,” on page 183

The chapters in this part are:

- Chapter 13, “First steps in problem determination,” on page 95
- Chapter 14, “AIX problem determination,” on page 97
- Chapter 15, “Linux problem determination,” on page 127
- Chapter 16, “Windows problem determination,” on page 143
- Chapter 17, “z/OS problem determination,” on page 151
- Chapter 18, “Sun Solaris problem determination,” on page 165
- Chapter 19, “Hewlett-Packard SDK problem determination,” on page 167
- Chapter 20, “ORB problem determination,” on page 169
- Chapter 21, “NLS problem determination,” on page 183



---

## Chapter 13. First steps in problem determination

Ask these questions before going any further:

**Have you enabled core dumps?**

Core dumps are essential to enable IBM Service to debug a problem. Although core dumps are enabled by default for the Java process (see Chapter 23, "Using dump agents," on page 195 for details), operating system settings might also need to be in place to allow 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 chapter for the platform.

**Can you reproduce the problem with the latest Service Refresh?**

The problem might also have been fixed in a recent service refresh. Make sure you are using the latest service refresh.

**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 platform specific, "Setting up and checking environment" sections in chapters Chapter 14 through Chapter 17).

**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 could be related to configuration of the JVM in a larger environment and might have been solved already in a Fix Pack. The problem could be related to native code executed by the JVM on behalf of other software. If this is so, the issue might have been resolved in a later version of any relevant software, for example DB2 or the WebSphere Application Server.

**Is the problem reproducible on the same machine?**

Knowing that this defect occurs every time the described steps are taken, is one of the most helpful things you can know about it and tends to indicate a straightforward programming error. If, however, it occurs at alternate times, or at one time in ten or a hundred, thread interaction and timing problems in general would be much more likely.

**Is the problem reproducible on another machine?**

A problem that is not evident on another machine could help you find the cause. A difference in hardware could 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 appear on multiple platforms?**

If the problem appears only on one platform, it could be related to a platform-specific part of the JVM or native code used within a user application. If the problem occurs on multiple platforms, the problem could be related to the user Java application or a cross-platform part of the JVM; for

## **First steps in problem determination**

example, Java Swing API. Some problems might be evident only on particular hardware; for example, Intel32. A problem on particular hardware could possibly indicate a JIT problem.

### **Does turning off the JIT help?**

If turning off the JIT prevents the problem, there might be a problem with the JIT. This can also indicate a race condition within the user Java application which 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 29, "JIT problem determination," on page 243.)

### **Have you tried reinstalling the JVM or other software and rebuilding relevant application files?**

Some problems occur from a damaged or invalid installation of the JVM or other software. It is also possible that an application could have inconsistent versions of binary files or packages. Inconsistency is particularly likely in a development or testing environment and could potentially be solved by getting a completely 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.

---

## Chapter 14. AIX problem determination

This chapter describes problem determination on AIX in:

- “Setting up and checking your AIX environment”
- “General debugging techniques” on page 99
- “Diagnosing crashes” on page 108
- “Debugging hangs” on page 109
- “Understanding memory usage” on page 112
- “Debugging performance problems” on page 119
- “I/O bottlenecks” on page 125
- “Collecting data from a fault condition in AIX” on page 125
- “Getting AIX technical support” on page 126

---

### 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 have to 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. Ensure that all the JVM files have installed in the correct location and that the correct permissions are set. The default installation directory is in /usr/java5 for the 32-bit JVM and /usr/java5\_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 contains the correct Java executable, or that the application you are using is pointing to the correct Java executable. 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 an install using smitty or installp. You can find the list of prerequisites in the *User Guide* that is shipped with the SDK. Use `ls1pp -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 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

## AIX - setting up and checking your environment

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 143. The ReportEnv tool is available on request from jvmcookbook@uk.ibm.com.

## Enabling full AIX core files

When a failure occurs, the most important diagnostic data to obtain is the process core file. The majority of the JVM settings are suitable by default, but, to ensure that this file is generated for the JVM on AIX, you must make a number of operating system settings.

### Operating system settings

1. To obtain full core files, 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 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. You can disable the generation of core files by using this Java command-line option:  
**-Xdump:system:none**

2. Set the following in SMIT:

- a. Start smit by typing smit as root
- b. Go to **System Environments -> Change/Show Characteristics of Operating System**
- c. Set the **Enable full CORE dump** option to **TRUE**

Alternatively, you can run chdev -l sys0 -a fullcore='true' as root.

You can check the setting with \$ lsattr -D -c sys -a fullcore -H, which should produce output similar to this:

```
attribute deflt description          user_settable
      fullcore  false Enable full CORE dump True
```

### 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 core file when a failure occurs, run the following:

```
java -Xdump:what
```

which should produce 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=nodumps
```

The values above are the default settings. At least events=gpf must be set to generate a core file when a failure occurs.

You can change and set options using the command-line option **-Xdump**, which is described in Chapter 23, "Using dump agents," on page 195.

### Available disk space

You must ensure that the disk space available is sufficient for the core file to be written to it. The JVM allows the core file to be written to any directory that is specified in the label option. For the core file to be written to this location, you must have sufficient disk space (up to 2 GB might be required for 32-bit and over 6 GB for 64-bit), and the correct permissions for the Java process to write to that location

---

## General debugging techniques

Below is a short guide to the JVM provided diagnostic tools and AIX commands that can be useful when diagnosing problems with the AIX JVM. In addition to the information given below, you can obtain AIX publications from the IBM Web site. Go to <http://www.ibm.com/servers/aix/>, click **Library**, and then choose the documentation link for your platform. Of particular interest are:

- Performance management and tuning
- Programming for AIX

You might also find this Redbook helpful: "*C and C++ Application Development on AIX*" (SG24-5674) , available from: <http://www.redbooks.ibm.com>.

### Starting Javadumps in AIX

See Chapter 24, "Using Javadump," on page 203.

### Starting Heapdumps in AIX

See Chapter 25, "Using Heapdump," on page 213.

## AIX debugging commands

### **bindprocessor -q**

This command shows how many processors are enabled.

### **bootinfo -K**

This command shows if the 64-bit kernel is active.

### **bootinfo -y**

This command shows whether the hardware in use is 32-bit or 64-bit.

### **dbx**

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

The Java 5.0 SDK also includes a dbx Plug-in for additional help debugging Java applications. See "DBX Plug-in" on page 107 for more information.

## AIX - general debugging techniques

### iostat

Use this command to determine if a system has an I/O bottleneck. The read and write rate to all disks is reported. This tool is useful in determining if you need to 'spread out' the disk workload across multiple disks. The tool, also reports the same CPU activity that **vmstat** does.

### lsattr

This command details characteristics and values for devices in the system. To obtain the processor type, and therefore the speed, use:

```
# lsattr -El proc0
state      enable      Processor state False
type      PowerPC_POWER3 Processor type  False
frequency 2000000000    Processor Speed False
```

### netpmmon

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,

```
netpmmon -o /tmp/netpmmon.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 netpmmon.

Process CPU Usage Statistics:

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

The adapter usage is shown here:

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

The following 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.

```
netpmmon -O so -t -o /tmp/netpmmon_so_thread.txt; sleep 20; trcstop
```

The extract below 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:

```
netpmmon -O so -o /tmp/netpmmon_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 turn on 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 as 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
```

To see the size of the wall use:

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

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

## AIX - general debugging techniques

### nmon

Nmon is a free software tool that 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 [www.ibm.com/servers/esdd/articles/analyze\\_aix/](http://www.ibm.com/servers/esdd/articles/analyze_aix/). The information that is collected includes CPU, disk, network, adapter statistics, kernel counters, memory, and the 'top' process information.

### ps

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 invoke **ps** to monitor 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 user ID 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 CPU usage across multiple CPU's. In this example below, two samples are taken every five seconds on a 2-processor system that is 80% utilized.

```
# 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

```

## AIX - general debugging techniques

1	35	42	0	22
-	33	45	0	22
Average	0	32	47	0
1	34	45	0	22
-	33	46	0	21

### svmon

Svmon captures snapshots of virtual memory. Using svmon to take snapshots of the memory usage of a process over regular intervals allows you to monitor its memory usage and check for unbounded memory growth that would be indicative of a memory leak. 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
2c7ea	3	work	shmat/mmap	36678	0	0	36656
3c80e	4	work	shmat/mmap	7956	0	0	7956
5cd36	5	work	shmat/mmap	7946	0	0	7946
14e04	6	work	shmat/mmap	7151	0	0	7151
7001c	d	work	shared library text	6781	0	0	736
0	0	work	kernel seg	4218	1552	182	3602
							0..22017 : 65474..65535
6cb5a	7	work	shmat/mmap	2157	0	0	2157
48733	c	work	shmat/mmap	1244	0	0	1244
cac3	-	pers	/dev/hd2:176297	1159	0	-	-
54bb5	-	pers	/dev/hd2:176307	473	0	-	-
78b9e	-	pers	/dev/hd2:176301	454	0	-	-
58bb6	-	pers	/dev/hd2:176308	254	0	-	-
cee2	-	work		246	17	0	246
4ccb3	-	pers	/dev/hd2:176305	226	0	-	-
7881e	-	pers	/dev/e2axa702-1:2048	186	0	-	-
68f5b	-	pers	/dev/e2axa702-1:2048	185	0	-	-
28b8a	-	pers	/dev/hd2:176299	119	0	-	-
108c4	-	pers	/dev/e2axa702-1:1843	109	0	-	-
24b68	f	work	shared library data	97	0	0	78
64bb9	-	pers	/dev/hd2:176311	93	0	-	-
74bbd	-	pers	/dev/hd2:176315	68	0	-	-
3082d	2	work	process private	68	1	0	68
10bc4	-	pers	/dev/hd2:176322	63	0	-	-
50815	1	pers	code,/dev/hd2:210969	9	0	-	-
44bb1	-	pers	/dev/hd2:176303	7	0	-	-
7c83e	-	pers	/dev/e2axa702-1:2048	4	0	-	-
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.

**cInt** 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.

**tprof**

Tprof is one of the AIX legacy tools that provides a detailed profile of CPU usage for every AIX process ID and name. There are more details on special Java options under profiling tools below.

**topas**

Topas is a useful graphical interface that will give you immediate information about system activity. The screen looks like this:

Topas Monitor for host: aix4prt	EVENTS/QUEUES	FILE/TTY
Mon Apr 16 16:16:50 2001	Interval: 2	Cswitch 5984 Readch 4864
		Syscall 15776 Writech 34280

## AIX - general debugging techniques

Kernel	63.1	#####						Reads	8	Rawin	0
User	36.8	#####						Writes	2469	Ttyout	0
Wait	0.0							Forks	0	Igets	0
Idle	0.0							Execs	0	Namei	4
								Runqueue	11.5	Dirblk	0
Network	KBPS	I-Pack	O-Pack	KB-In	KB-Out		Waitqueue	0.0			
lo0	213.9	2154.2	2153.7	107.0	106.9						
tr0	34.7	16.9	34.4	0.9	33.8	PAGING			MEMORY		
Disk	Busy%	KBPS	TPS	KB-Read	KB-Writ		Faults	3862	Real,MB	1023	
hdisk0	0.0	0.0	0.0	0.0	0.0		Steals	1580	% Comp	27.0	
							PgspIn	0	% Noncomp	73.9	
							PgspOut	0	% Client	0.5	
Name	PID	CPU%	PgSp	Owner			PageIn	0			
java	16684	83.6	35.1	root			PageOut	0	PAGING SPACE		
java	12192	12.7	86.2	root			Sios	0	Size,MB	512	
lprud	1032	2.7	0.0	root					% Used	1.2	
aixterm	19502	0.5	0.7	root			NFS (calls/sec)	% Free	98.7		
topas	6908	0.5	0.8	root			ServerV2	0			
ksh	18148	0.0	0.7	root			ClientV2	0	Press:		
gil	1806	0.0	0.0	root			ServerV3	0	"h" for help		

### trace

This command captures a sequential flow of time-stamped system events. The trace is a valuable tool for observing system and application execution. While many of the other tools provide high level statistics such as CPU and I/O utilization, the trace facility helps expand the information about where the events happened, which process is responsible, when the events took place, and how they are 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 is used to analyze lock activity in the AIX kernel and kernel extensions for simple locks.

### truss

This command traces a process's system calls, dynamically loaded user-level function calls, received signals, and incurred machine faults.

### 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
```

The first line of information returned is the time since system reboot, and is normally ignored.

A typical output looks like this:

kthr	memory				page				faults				cpu				time			
r	b	avm	fre	re	pi	po	fr	sr	cy	in	sy	cs	us	sy	id	wa	hr	mi	se	
0	0	45483	221	0	0	0	0	1	0	224	326	362	24	7	69	0	15:10:22			
0	0	45483	220	0	0	0	0	0	0	159	83	53	1	1	98	0	15:10:23			
2	0	45483	220	0	0	0	0	0	0	145	115	46	0	9	90	1	15:10:24			

In this output, look for:

- Columns r (run queue) and b (blocked) starting to go up, especially above 10. This rise usually indicates that you have too many processes competing for CPU.
- Values in the pi, po (page in/out) columns at non-zero, possibly indicating that you are paging and need more memory. It might be possible that you have the stack size set too high for some of your JVM instances.
- cs (contact switches) going very high compared to the number of processes. You might need 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

A crash can occur only 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 third-party 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 normal process.

If a crash occurs, you should gather some basic documents. These documents either point to the problem that is in the application or third party package JNI code, or help the IBM JVM Support team to diagnose the fault.

### Documents to gather

When a crash takes place, the following diagnostic data is required to help diagnose the problem:

- The output of stackit.sh run against the core file located as specified by the `label` field of **-Xdump:what**. stackit.sh is a script that runs a dbx session and is available from your Support Representative or from [jvmcookbook@uk.ibm.com](mailto:jvmcookbook@uk.ibm.com).

- The output of JExtract run against the core file:

```
jextract [core file]
```

and collect the `core.{date}.{time}.{pid}.dmp.zip` output. See Chapter 28, “Using the dump formatter,” on page 225 for details about jextract.

- Collect the javadump file. This file should be in the location detailed against the `label` field in **-Xdump:what** for javadumps

- Collect any `stdout` and `stderr` output generated by the Java process

- Collect the system error report:

```
errpt -a > errpt.out
```

The above steps should leave you with the following files:

- `stackit.out`
- `core.{date}.{time}.{pid}.dmp.zip`
- `javacore.{date}.{time}.{pid}.txt`
- `errpt.out`
- `stderr/stdout` files

### Interpreting the stack trace

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 29, “JIT problem determination,” on page 243). 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 third party JNI code.

## 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

For an explanation of deadlocks and how the Javadump tool is used to diagnose them, see “Locks, monitors, and deadlocks (LOCKS)” on page 207.

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 99. For example:

## AIX - debugging hangs

```
$ 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.

## Investigating busy hangs in AIX

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 (for example, sysRecv) 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?

If the process seems still 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 's to the *k-tid* value from the dbx **thread** command, it can be seen that the currently running methods in this case are reverseHandle and getLinkRegister.

Now you can use **dbx** to generate the C thread stack for these two threads using the dbx **thread** command for the corresponding dbx thread numbers (\$tx). To obtain the full stack trace including Java frames, map the dbx thread number to the threads *pthread\_t* value, which is listed by the Javadump file, and can be obtained from the ExecEnv structure for each thread using the Dump Formatter. Do this with the **dbx** command **thread info [dbx thread number]**, which produces an output of the form:

```
thread state-k      wchan      state-u      k-tid      mode held scope function
$ t51    run          running      43939      u    no  sys  reverseHandle
general:
  pthread addr = 0x220c2dc0      size      = 0x18c
  vp addr     = 0x22109f94      size      = 0x284
  thread errno = 61
  start pc     = 0xf04b4e64
  joinable     = yes
  pthread_t     = 3233
scheduler:
  kernel      =
  user        = 1 (other)
event :
  event       = 0x0
  cancel      = enabled, deferred, not pending
stack storage:
  base        = 0x220c8018      size      = 0x40000
  limit       = 0x22108018
  sp          = 0x22106930
```

Showing that the TID value from **ps** (**k-tid** in **dbx**) corresponds to dbx thread number 51, which has a *pthread\_t* of 3233. Looking for the *pthread\_t* in the Javadump file, you now have a full stack trace:

```
"Worker#31" (TID:0x36288b10, sys_thread_t:0x220c2db8) Native Thread State:
ThreadID: 00003233 Reuse: 1 USER SUSPENDED Native Stack Data : base: 22107f80
pointer 22106390 used(7152) free(250896)
----- Monitors held -----
java.io.OutputStreamWriter@3636a930
com.ibm.servlet.engine.webapp.BufferedWriter@3636be78
com.ibm.servlet.engine.webapp.WebAppRequestDispatcher@3636c270
com.ibm.servlet.engine.srt.SRTOutputStream@36941820
com.ibm.servlet.engine.oselistener.nativeEntry.NativeServerConnection@36d84490 JNI pinning lock

----- Native stack -----
_spin_lock_global_common pthread_mutex_lock - blocked on Heap Lock
sysMonitorEnterQuicker sysMonitorEnter unpin_object unpinObj
jni_ReleaseScalarArrayElements jni_ReleaseByteArrayElements
Java_com_ibm_servlet_engine_oselistener_nativeEntry_NativeServerConnection_nativeWrite

----- Java stack ----- () prio=5

com.ibm.servlet.engine.oselistener.nativeEntry.NativeServerConnection.write(Compiled Code)
com.ibm.servlet.engine.srp.SRPConnection.write(Compiled Code)
```

## AIX - debugging hangs

```
com.ibm.servlet.engine.srt.SRTOutputStream.write(Compiled Code)
java.io.OutputStreamWriter.flushBuffer(Compiled Code)
java.io.OutputStreamWriter.flush(Compiled Code)
java.io.PrintWriter.flush(Compiled Code)
com.ibm.servlet.engine.webapp.BufferedWriter.flushChars(Compiled Code)
com.ibm.servlet.engine.webapp.BufferedWriter.write(Compiled Code)
java.io.Writer.write(Compiled Code)
java.io.PrintWriter.write(Compiled Code)
java.io.PrintWriter.write(Compiled Code)
java.io.PrintWriter.print(Compiled Code)
java.io.PrintWriter.println(Compiled Code)
pagecompile._identifycustomer_xjsp.service(Compiled Code)
javax.servlet.http.HttpServlet.service(Compiled Code)
com.ibm.servlet.jsp.http.pagecompile.JSPState.service(Compiled Code)
com.ibm.servlet.jsp.http.pagecompile.PageCompileServlet.doService(Compiled Code)
com.ibm.servlet.jsp.http.pagecompile.PageCompileServlet doGet(Compiled Code)
javax.servlet.http.HttpServlet.service(Compiled Code)
javax.servlet.http.HttpServlet.service(Compiled Code)
```

And, using the full stack trace, it should be possible to identify any infinite loop that might be occurring. The 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.

---

## 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. Process addressability to data is managed at the segment level, so a data segment can either be shared (between processes), or private.

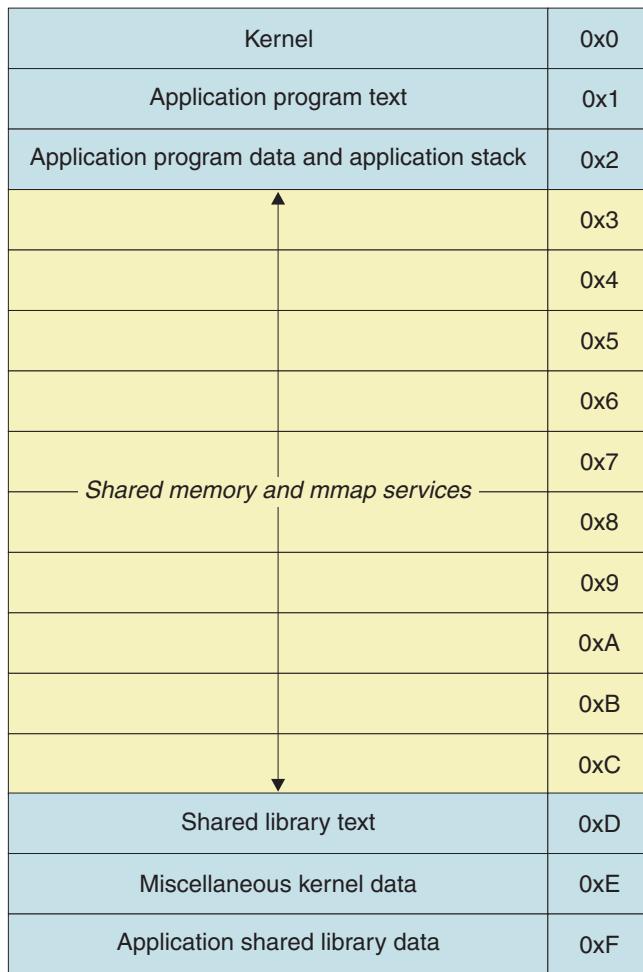


Figure 11. The AIX 32-Bit Memory Model with MAXDATA=0 (default)

- 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.
- Segments D and F are shared library text and data areas respectively.
- Segment E is also shared memory and miscellaneous kernel usage.

## The 64-bit AIX Virtual Memory Model

The 64-bit model allows many more segments, although each segment is still 256 MB. Again, addressability is managed at segment level, but the granularity of function for each segment is much finer.

With the greater addressability 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 chapter, 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.

### 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 0xFFFFFFFF, the process will not use global shared libraries, but load them privately. So 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.

Further details of the AIX Memory Models can be found at: [http://publib.boulder.ibm.com/infocenter/pseries/....](http://publib.boulder.ibm.com/infocenter/pseries/)

### 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, are maintained by different mechanisms, and are largely independent of each other.

The Java heap contains the instances of Java objects and is often referred to simply 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, "Understanding the Garbage Collector," 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 Inflaters and Deflators, malloc allocations by application JNI code, compiled code generated by the Just In Time (JIT) Compiler, and 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 in separation 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	Pgsp	Virtual	64-bit	Mthrd	LPage		
29670	java	87347	4782	5181	95830		N	Y		N
Vsid	Esid	Type	Description			LPage	Inuse	Pin	Pgsp	Virtual
50e9		- work				-	41382	0	0	41382
			Addr Range: 0..41381							
9dfb		- work				-	28170	0	2550	30720
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							

## AIX - understanding memory usage

1140	6 work other segments Addr Range: 0..32780	-	26	0	0	26
cdf1	- work Addr Range: 0..5277	-	2	0	0	2
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	- cInt /dev/fslv00:44722 Addr Range: 0..207	-	0	0	-	-
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 largely 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(1)C
+ (warm) java/lang/String.charAt(1)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, but for most applications the performance gains are likely to be small. It will be of particular benefit to any application that makes heavy use of malloc calls in native code. For more information, see this article:<http://www-128.ibm.com/developerworks/eserver/library/es-appdev-aix5l.html>.

## 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. Turn on garbage collection's verbose tracing using the command-line option **-verbose:gc** to cause a report to be written to stderr each time garbage collection occurs. You can also direct this output to a log file using:

```
-Xverboselog:[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 30, “Garbage Collector diagnostics,” on page 247 for more information on verbose gc output and monitoring.

## Receiving OutOfMemory errors

Any OutOfMemory condition that occurs could be the result of either running out of Java heap or 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, it is entirely possible that there is not a memory leak as such, just that the steady state of memory usage 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 will not solve the problem, but will delay the onset of the OutOfMemory or error conditions, which can be of help on any production system.

The 32-bit JVM has these limits:

- The maximum size of an object that can be created is 1 GB.
- For an array object, the maximum number of array elements supported is  $(2^{28} - 1)$ . So, for a byte array, the maximum size of an array object is 256 MB.

## 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, consulting Appendix C, “Messages,” on page 345 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 **-verbose:gc** as detailed above, and the native heap 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, “Understanding the Garbage Collector,” 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, “Understanding the Garbage Collector,” 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 25, “Using Heapdump,” on page 213. Similar information can be gained by using other tools, such as JProbe and OptmizeIt.

## Native heap exhaustion

You can identify native heap exhaustion by monitoring the svmon snapshot output as discussed above. 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 improve the profiling.

In the case of DB2, 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 occur also without the Inuse pages approaching 65535 Inuse pages. This can be caused by fragmentation of the AIX malloc heaps, which is how AIX handles the native heap of the JVM.

This kind of OutOfMemory condition can again be identified from the svmon snapshots. Whereas previously the important column to look at for a memory leak is the Inuse values, 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. This 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 thereby solve problems of this type:

### MALLOCTYPE=3.1

This option allows 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** allows users to 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 not recommended

#### **MALLOCTYPE=buckets**

Malloc buckets provides 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 is enabled, allocation requests that fall within 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:** The above options might cause a percentage of performance hit. 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: pathname of file for malloc statistics>**

See above.

## **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 79, and the data that should be included in the bug report is listed below:

- 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

## AIX - debugging performance problems

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	0	114	4573	122	96	4	0	0
1	0	189898	611	0	0	0	0	0	0	0	115	420	102	99	0	0	0
1	0	189898	611	0	0	0	0	0	0	0	115	425	91	99	0	0	0
1	0	189898	611	0	0	0	0	0	0	0	114	428	90	99	0	0	0
1	0	189898	610	0	1	0	0	0	0	0	117	333	102	97	3	0	0
1	0	189898	610	0	0	0	0	0	0	0	114	433	91	99	1	0	0
1	0	189898	610	0	0	0	0	0	0	0	114	429	94	99	1	0	0
1	0	189898	610	0	0	0	0	0	0	0	115	437	94	99	0	0	0
1	0	189898	609	0	1	0	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 99.

### 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
wait		4	40.33	40.33	0.00	0.00
/usr/bin/tprof		1	0.20	0.02	0.00	0.18
/etc/syncd		3	0.05	0.05	0.00	0.00
/usr/bin/sh		2	0.01	0.00	0.00	0.00
gil		2	0.01	0.01	0.00	0.00
afsd		1	0.00	0.00	0.00	0.00
rpc.lockd		1	0.00	0.00	0.00	0.00
swapper		1	0.00	0.00	0.00	0.00
	====	=====	=====	====	=====	====
Total		20	100.00	64.70	0.00	35.29

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

## AIX - debugging performance problems

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/libpthread.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:

#### libbcv23.so

Bytecode Verifier

#### libdbg23.so

Debug Server (used by the Java Debug Interface)

#### libj9gc23.so

Garbage Collection

#### libj9extract.so

The dump extractor, used by the jextract command

#### libj9jit23.so

The Just In Time (JIT) Compiler

#### libj9jpi23.so

The JVMPPI interface

#### 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 zip utility library
| libjclsae_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
lelTask:::handleNextWorkUnit(MM_EnvironmentModron*) 1.05 ParallelTask.cpp
orkPackets:::getPacket(MM_Environment*,MM_Packet**) 0.70 WorkPackets.cpp
cheme:::fixupRegion(J9Object*,J9Object*,bool,long&) 0.67 CompactScheme.cpp
WorkPackets:::putPacket(MM_Environment*,MM_Packet*) 0.47 WorkPackets.cpp
rkingScheme:::scanObject(MM_Environment*,J9Object*) 0.43 MarkingScheme.cpp
sweepChunk(MM_Environment*,MM_ParallelSweepChunk*)
ment*,J9IndexableObject*,J9Object**,unsigned long) 0.42 allelSweepScheme.cpp
M_CompactScheme:::getForwardingPtr(J9Object*) const 0.38 MarkingScheme.cpp
ObjectHeapIteratorAddress0OrderedList:::nextObject() 0.36 CompactScheme.cpp
ckets:::getInputPacketFromOverflow(MM_Environment*)
.MM_WorkStack:::popNoWait(MM_Environment*) 0.33 dressOrderedList.cpp
WorkPackets:::getInputPacketNowait(MM_Environment*) 0.32 WorkPackets.cpp
canReferenceMixedObject(MM_Environment*,J9Object*) 0.29 WorkPackets.cpp
MarkingScheme:::markClass(MM_Environment*,J9Class*)
._ptrgl 0.29 MarkingScheme.cpp
MarkingScheme:::initializeMarkMap(MM_Environment*) 0.27 MarkingScheme.cpp
.MM_HeapVirtualMemory:::getHeapBase() 0.26 ptrgl.s
                                            0.25 MarkingScheme.cpp
                                            0.23 eapVirtualMemory.cpp

```

This output shows that the most-used functions are:

```

MarkingScheme:::scanMixedObject(MM_Environment*,J9Object*) 2.67 MarkingScheme.cpp
MarkingScheme:::scanClass(MM_Environment*,J9Class*) 2.54 MarkingScheme.cpp
ObjectSlotIterator.GC_ConstantPoolObjectSlotIterator:::nextSlot() 1.96 ObjectSlotIterator.cpp
ParallelTask:::handleNextWorkUnit(MM_EnvironmentModron*) 1.05 ParallelTask.cpp

```

The values show that the time is being spent during the Mark phase of GC. Because the output also contains references to the Compact and Sweep phases, it is

## AIX - debugging performance problems

likely that GC is 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 javacore.txt file (or a series of javacore.txt files) 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 javacore.txt 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 javacore.txt file.

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 javacore.txt file:

```
3XMTHREADINFO      "main" (TID:0x300E3500, sys_thread_t:0x30010734,  
                         state:R, native ID:0x000C8075) prio=5  
4XESTACKTRACE      at java/lang/Runtime.gc(Native Method)  
4XESTACKTRACE      at java/lang/System.gc(System.java:274)  
4XESTACKTRACE      at GCTest.main(GCTest.java:5)
```

These entries show that, in this case, the thread is calling GC, and explains the time spent in the 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 memory ( $144805 * 4 \text{ KB} = 565.64 \text{ MB}$ ). 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 **vmtune** command to alter the **maxperm** and **minperm** values.

If the Java process continues to increase its memory usage, an eventual memory constraint will be caused by a memory leak.

## I/O bottlenecks

This book does not discuss conditions in which the system is disk- 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. See "How to do heap sizing" on page 20.

## 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 compilation overhead. 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, "Understanding the JIT," on page 35 and Chapter 29, "JIT problem determination," on page 243.

## 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 35, "Using the HPROF Profiler," on page 319. **-Xrunhprof:help** gives you a list of suboptions that you can use with hprof.

---

## Collecting data from a fault condition in 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 97), 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**.

## AIX - collecting data from a fault condition

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 Formatter

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, or a canned dbx session. dbx does not, however, have the advantage of understanding Java frames and the JVM control blocks that the Dump Formatter does. Therefore, you are recommended to use the Dump Formatter in preference to dbx.

### The JavaDump file:

When a Javadump is written, a message (JVMDUMP010I) is written to stderr telling you the name and full path of the Javadump file. In addition, a Javadump file can be actively generated from a running Java process by sending it a **SIGQUIT (kill -3 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 would otherwise be ambiguous.

---

## Getting AIX technical support

See these web pages:

<http://techsupport.services.ibm.com/server/nav?fetch=a4ojc>  
<http://techsupport.services.ibm.com/server/nav?fetch=a5oj>

---

## Chapter 15. Linux problem determination

This chapter describes problem determination on Linux in:

- “Setting up and checking your Linux environment”
- “General debugging techniques” on page 128
- “Diagnosing crashes” on page 134
- “Debugging hangs” on page 135
- “Debugging memory leaks” on page 135
- “Debugging performance problems” on page 136
- “Collecting data from a fault condition in Linux” on page 138
- “Known limitations on Linux” on page 140

---

### Setting up and checking your Linux environment

**Note:** Linux operating systems undergo a large number of patches and updates. It is impossible for IBM personnel to 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 143. The ReportEnv tool is available on request from [jvmcookbook@uk.ibm.com](mailto: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 core files

When a crash occurs, the most important diagnostic data to obtain is the process core file. To ensure that this file is generated for the JVM on Linux, you make a number of settings. Most JVM settings are defaults, but operating system settings must also be correct, and these vary by distribution and Linux version. The following settings must be in place.

### Operating system settings

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 -u unlimited      sets the user memory limit to unlimited  
ulimit -f unlimited      sets the file limit to unlimited
```

## Linux - setting up and checking your environment

The current ulimit settings can be displayed using:

```
ulimit -a
```

These values are the "soft" limit, and are done 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 Version 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=/u/cbailey/sdk/jre/bin/core.%Y%m%d.%H%M%S.%pid.dmp
range=1..0
priority=999
request=nodumps
```

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 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=/u/cbailey/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 even larger for a 64-bit process), and the correct permissions for the Java process to write to that location.

## Threading libraries

The distributions supported by the IBM v5.0 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 dependency of your application.

---

## 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.

## Starting Javadumps in Linux

See Chapter 24, "Using Javadump," on page 203.

## Starting heapdumps in Linux

See Chapter 25, "Using Heapdump," on page 213.

## Using the dump extractor on Linux

When a core dump occurs, the following items are required so that you can analyze the core file:

- 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, run the jextract utility against the core file:

```
jextract <core file name>
```

to generate a file called dumpfilename.zip in the current directory.

dumpfilename.zip is a compressed file containing the required files. Running jextract against the core file also allows for the subsequent use of the Dump Formatter.

## Using core dumps

The commands **objdump** and **nm** both display information about object files. If a crash occurs and a corefile is produced, these commands help you analyze the file.

### **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 invoke objdump, type: **objdump <option> <filename>**

### **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> <filename>**

You can see a complete list of options by typing **objdump -H**. The **-d** option disassembles contents of executable sections

Run these commands on the same machine as the one that produced the core files to get the most accurate symbolic information available. This output (together with the core file, if small enough) is used by IBM Java Support to diagnose a problem.

## Using system logs

The kernel provides useful environment information. Use the following commands to view this information:

- **ps -elf**
- **top**
- **vmstat**

The **ps** command displays process status. Use it to gather information about native threads. Some useful options are:

- **-e**: Select all processes
- **-l**: Displays in long format

## Linux - general debugging techniques

- **-f**: Displays a full listing

The **top** command displays the most CPU- 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. The display is updated every five seconds by default, although this can be changed by using the **s** (interactive) command. 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 this down into tasks that are running, sleeping, stopped, or undead. In addition to displaying PID, PPID, and UID, the **top** command displays information on 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 reports virtual memory statistics. It is useful to perform a general health check on your system, although, because it reports on the system as a whole, commands such as **ps** and **top** can be used afterwards to gain more specific information about your programs operation. When you use it for the first time during a session, the information is reported as averages since the last reboot. However, further usage will display reports that are based on a sampling period that you can specify as an option. **Vmstat 3 4** will display values every 3 seconds for a count of 4 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 appears in five sections; processes, memory, swap, io, system, and cpu.

The **processes** section shows how many processes are awaiting run time, blocked, or swapped out.

The **memory** section 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.

The **swap** section shows the kilobytes per second of memory swapped in from and swapped out to disk. Memory is swapped out to disk if RAM is not big enough 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).

The **io** section shows the number of blocks per second of memory sent to and received from block devices.

The **system** section displays the interrupts and the context switches per second. There is overhead associated with each context switch so a high value for this may mean that the program does not scale well.

The **cpu** section shows a break down 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 very busy. You can use this knowledge to help you understand which areas of your program are using the CPU the most.

In Linux, each native thread is a distinct process with a unique process ID (PID). The kernel can therefore provide very useful information about your threads through commands such as **ps** and **top**.

## Linux debugging commands

### ps

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 filesystem. 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.
- 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 example above, 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.

### Tracing

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 active, 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 it with the **-f** option, it will trace child processes that have been created as a result of a forked system call. strace is often used to investigate plug-in problems or to try to understand why programs do not start properly.

#### 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.

#### 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

## Linux - general debugging techniques

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.

### 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 very 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.

You can run gdb in three ways:

#### Starting a program

Normally 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 invoke 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 executable as the first argument and the pid as the second argument:

```
gdb <Java Executable> <PID>
```

When gdb is attached to a running program, this program is halted and its position within 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 corefile

A corefile is normally produced when a program crashes. gdb can be run on this corefile. The corefile 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. With this information, you should be able to discover what caused the crash. To debug a corefile, invoke gdb with the Java executable as the first argument and the corefile name as the second argument:

```
gdb <Java Executable> <corefile>
```

When you run gdb against a corefile, it will initially show 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 your input and will continue in whichever way you choose.

There are a number of ways of controlling execution and examination of the code. Breakpoints can be set for a particular line or function using the command:

```
breakpoint linenumber
or
breakpoint functionName
```

After you have set a breakpoint, use the **continue** command to allow the program to execute until it hits a breakpoint.

Set breakpoints using conditionals so that the program will halt only when the specified condition is reached. For example, using **breakpoint 39 if var == value** causes the program to halt on line 39 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. Note that setting a breakpoint on 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 nowrap
handle sigusr2 pass nostop nowrap
```

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 (the most recently called function appears at the top of the call stack), so 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 to the very left, 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.

## Linux - general debugging techniques

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, but while **next** treats function calls as a single line of execution, **step** will step through each line of the called function.

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 datatype 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**

The **help** command displays a list of subjects, each of which can have the help command invoked 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 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 JVM to crash. For example, this could be a resource shortage (such as memory or disk) or a stability problem. Check the Javadump file, which contains various system information (as described in Chapter 24, “Using Javadump,” on page 203). The Javadump 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 28, “Using the dump formatter,” on page 225) 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 compiled code. If you have a problem with the JIT, try running with JIT off by setting 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 34, “Using the Reliability, Availability, and Serviceability Interface,” on page 305.

## Finding out about the Java environment

Use the Javadump 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 2, “Understanding the Garbage Collector,” on page 7.
- The crash occurred after garbage collection, indicating a possible memory corruption.

For more information about the Garbage Collector, see Chapter 2, “Understanding the Garbage Collector,” on page 7.

---

## Debugging hangs

For an explanation of deadlocks and diagnosing them using the Javadump tool, see “Locks, monitors, and deadlocks (LOCKS)” on page 207.

A hang is caused by a wait or a loop. A wait or deadlock sometimes occurs because of a wait on a lock or monitor. A loop or livelock can occur similarly or sometimes because of an algorithm making little or no progress towards completion. The following approaches are most useful in this situation:

- Monitoring process and system state (as described in “Collecting data from a fault condition in Linux” on page 138).
- Java Dumps give monitor and lock information.
- `verbose:gc` information is useful. It indicates:
  - Excessive garbage collection because of lack of Java heap space causing the system to appear to be in livelock
  - Garbage collection causing of 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 25, “Using Heapsdump,” on page 213.

## 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. First, 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. Although most of these tools have been introduced earlier in this chapter, it is still worth mentioning them here, and discussing how you can use them to specifically debug performance issues.

The aspects of the system that you are most interested in measuring are CPU usage and memory usage. If you can prove that the CPU is not powerful enough to handle the workload, any amount of tuning makes not much difference to overall performance. Nothing less than a CPU upgrade might be required.

Similarly, if a program is running in an environment in which it does not have enough memory, an increase in the memory is going to make a much bigger change to performance than any amount of tuning does.

### CPU usage

You might typically experience Java processes consuming 100% of processor time when a process reaches its resource limits. Ensure that **ulimit** settings are appropriate to the application requirement. See “Operating system settings” on page 127 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. Because Java threads are run as system processes, you can learn valuable information about the performance of your application. See /proc man for more information about viewing /proc information. /proc/version gives you information about the Linux kernel that is on your 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 “Using system logs” on page 129.

## 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 gives information on your swap file.

Swap space is used as an extension of the systems virtual memory. 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 so reduces the chance of further bottlenecks.

VMstat is a tool that enables you to discover where performance problems might be caused. For example, if you see that high swap rates are occurring, it is likely that you 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 it is quite likely that you need to increase your physical memory. More details on how to use VMstat are provided in "Using system logs" on page 129.

## Network problems

Another area that often affects performance is the network. Obviously, the more you know about the behavior of your program, the easier it is for you to decide whether this is a likely source of performance bottleneck. If you think that your program is likely to be I/O bound, netstat is a useful tool. In addition to providing information about network routes, netstat gives a list of active sockets for each network protocol and can give overall statistics, such as the number of packets that are received and sent. Using netstat, you can see how many sockets are in a CLOSE\_WAIT or ESTABLISHED state and you can tune the respective TCP/IP parameters accordingly for better performance of the system. For example, tuning /proc/sys/net/ipv4/tcp\_keepalive\_time will reduce the time for socket waits in TIMED\_WAIT state before closing a socket. If you are tuning /proc/sys/net file system, the effect will be on all the applications running on the system. However, to make a change to an individual socket or connection, you have to use Java Socket API calls (on the respective socket object). Use **netstat -p** (or the **lsof** command) to find the right PID of a particular socket connection and its stack trace from a javacore file taken with the **kill -3 <pid>** command.

You can also use IBM's RAS trace, **-Xtrace:print=net**, to trace out network-related activity within 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 underperforming because of TCP/IP problems. The more you understand your hardware capacity, the easier it is for you to tune with confidence the parameters of particular system components that will improve the overall performance of your application. You can also determine whether only system tuning and tweaking will noticeably improve performance, or whether actual upgrades are required.

### JVM heap sizing

The Java heap size is one of the most important tuning parameters of your JVM. See “How to do heap sizing” on page 20.

### 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 compilation overhead. 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, “Understanding the JIT,” on page 35 and Chapter 29, “JIT problem determination,” on page 243.

### 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 35, “Using the HPROF Profiler,” on page 319. `-Xrunhprof:help` gives you a list of suboptions that you can use with hprof.

The Performance Inspector<sup>TM</sup> package contains a suite of performance analysis tools for Linux. You can use tools to help identify performance problems in your application as well as to understand how your application interacts with the Linux kernel. See <http://perfinsp.sourceforge.net/> for details.

---

## Collecting data from a fault condition in 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 IBM Java Service solve the problem.

### Collecting core files

Collect corefiles to help diagnose many types of problem. Process the corefile with jextract. The resultant jar file is useful for service (see “jextract” on page 225).

### Producing Javadumps

In some conditions (a crash, for example), a Javadump 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 (`kill -3 <PID>`) signal. This is discussed in more detail in Chapter 24, “Using Javadump,” on page 203.

### 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

The following commands can be useful to determine the operating environment of a process at various stages of its lifecycle:

### **uname -a**

Provides operating system and hardware information.

### **df**

Displays free disk space on a system.

### **free**

Displays memory use information.

### **ps -ef**

Gives a full process list.

### **lsof**

Lists open file handles.

### **top**

Displays process information (such as processor, memory, states) sorted by default by processor usage.

### **vmstat**

Provides general memory and paging information.

In general, the **uname**, **df**, and **free** output is the most useful. The other commands may 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 whose name has 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 invoked 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/N directory contains detailed diagnostic information about the process with PID (process id) N, where N is the id of the process.

The command cat /proc/N/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" on page 131.

---

## Known limitations on Linux

### 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
- Hang

Before kernel 2.4, the maximum number of threads available is determined by the minimum of:

- The user processes setting (**ulimit -u**) in `/etc/security/limits.conf`.
- The limit **MAX\_TASKS\_PER\_USER** defined in `/usr/include/linux/tasks.h`. (This change requires the Linux kernel to be recompiled.)
- The limit **PTHREAD\_THREADS\_MAX** defined in `libpthread.so`. (This change requires the Linux kernel to be recompiled.)

However, you might run out of virtual storage before reaching the maximum number of threads.

In kernel 2.4, the native stack size is the main limitation when running a large number of 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. Thus, 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 128.)

### 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 a down-level 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 install 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*.

## **Linux - known limitations**

---

## Chapter 16. Windows problem determination

This chapter describes problem determination on Windows in:

- “Setting up and checking your Windows environment”
- “General debugging techniques” on page 145
- “Diagnosing crashes in Windows” on page 146
- “Debugging hangs” on page 147
- “Debugging memory leaks” on page 148
- “Debugging performance problems” on page 149
- “Collecting data from a fault condition in Windows” on page 150

---

### Setting up and checking your Windows environment

The operation of the JVM 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.

A known problem for first-time users is to install Java and then set up a work directory and compile a ‘Hello World’ program. If you cannot run HelloWorld, possibly the **CLASSPATH** variable is not pointing to your .CLASS file. A solution is to type set **CLASSPATH=.**, which always allows you to find classes in your current directory.

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.” The ReportEnv tool is available on request from [jvmcookbook@uk.ibm.com](mailto:jvmcookbook@uk.ibm.com).

Figure 12 on page 144 shows the ReportEnv tool.

## Windows - setting up and checking your environment

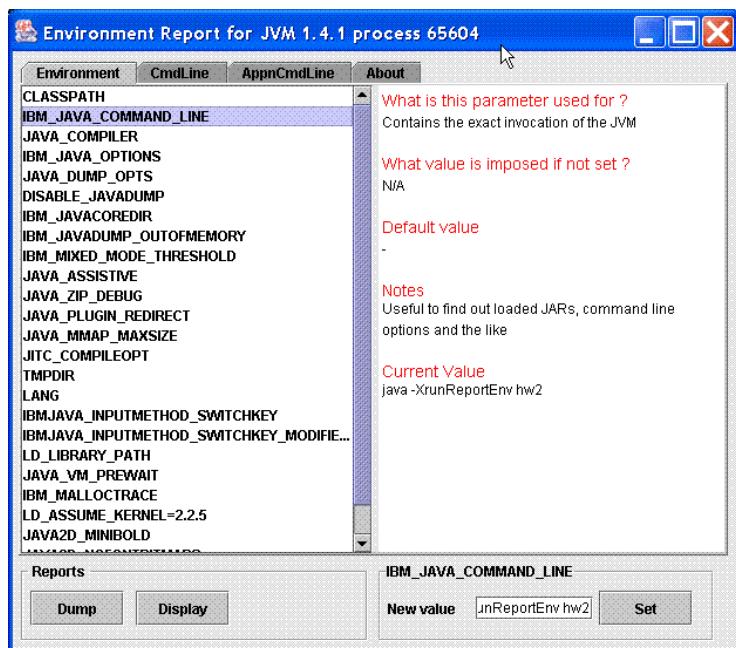


Figure 12. Screenshot of the ReportEnv tool

## Setting up your Windows environment for data collection

### Setting up for dump extraction

To enable the JVM to generate a dump for use by the cross platform debugger, see Chapter 28, "Using the dump formatter," on page 225.

### Setting up for Javadump and Heapdump

Refer to Chapter 24, "Using Javadump," on page 203 and Chapter 25, "Using Heapdump," on page 213.

### Native Windows tools

**Generating a user dump file in a hang condition:** Windows provides a facility that generates a user dump file for any process (even if it is hung) through a utility called userdump.exe. This utility is provided by Microsoft and you can download it from their Web site: [www.microsoft.com](http://www.microsoft.com).

Usage:

**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 from where userdump.exe is run).

For more information about generating a user dump file in a hang condition, see "Debugging hangs" on page 147.

## General debugging techniques

This section provides a guide to the JVM-provided diagnostic tools and Windows commands that can be useful when you are diagnosing problems that occur with the Windows JVM.

### Starting Javadumps in Windows

See Chapter 24, "Using Javadump," on page 203.

### Starting Heapdumps in Windows

See Chapter 25, "Using Heapdump," on page 213.

### Using the Cross-Platform Dump Formatter

The IBM Java Cross-Platform Dump Formatter is a powerful tool for debugging many fault scenarios. As the name implies, it is a cross-platform tool and takes its input from a predefined data source or code plug-in. The data source must be generated by platform code because crash dumps vary according to the architecture. See Chapter 28, "Using the dump formatter," on page 225 for details.

### 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_1HdSMR (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 descending order):

## Windows - general debugging techniques

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

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 usually refers to java.exe as the application that caused the crash. Crashes can occur because of a fault in the JVM, 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 JVM. Otherwise, the fault must be in other code. Try and find out which is the case so that you can pinpoint a problem.

As a general rule, try to recreate 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 Java service team. You should:

- Collect the Javadump. See Chapter 24, “Using Javadump,” on page 203 for details.
- Collect the crash dump. See “Setting up and checking your Windows environment” on page 143 for details.
- Collect the snap trace file. See Chapter 33, “Tracing Java applications and the JVM,” on page 283 for details.
- Run with the JIT turned off. See Chapter 29, “JIT problem determination,” on page 243 for details.
- If the problem occurs with or without the JIT, specify the JVM option **-Xjit:count=n**. **-Xjit:count=n** specifies the number of times a method is invoked before it is compiled. This way, the JVM starts up reasonably quickly (there is no overhead of “JITting” all the basic methods) and keeps the advantages of having a JIT. During the default JIT operation, some methods in your code are interpreted and some are executed as native code, depending on whether they have hit the threshold. If you pass the option **-Xjit:count=0** to the JVM, the JIT starts “JITting” all methods (that is, no code is interpreted). Run your application with **-Xjit:count=0** and collect the Javadump and the log. This is the opposite of the previous scenario where no code was “JIT’d”.
- Collect the Javadump log if the problem still occurs.
- Try some JIT compile options. If the problem disappears with the JIT turned off, try some JIT compile options to see if the problem can be narrowed down further. You could find that you can continue using the JVM, albeit with reduced JIT performance, while giving the service team a running start with your bug report. For information on using the basic JIT compile options, see Chapter 29, “JIT problem determination,” on page 243.

- Try adjusting the garbage collection parameters. See Chapter 2, “Understanding the Garbage Collector,” on page 7 for details. Make a note of any changes in behavior.
- Try running on a uniprocessor box. If your problem is occurring on a multiprocessor system, test your application on a uniprocessor box. 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 them appropriately and make them available to IBM. (See Part 2, “Submitting problem reports,” on page 79 for details.) The required files are:

- The JVM-produced Javadump file (Javacore)
- 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% 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.

## Analyzing deadlocks

For an explanation of deadlocks and diagnosing them using the Javadump tool, see “Locks, monitors, and deadlocks (LOCKS)” on page 207.

## Getting a dump from a hung JVM

The Windows JVM is configured to do a dump extraction if it terminates abnormally. Also, you can cause a dump by configuring the JVM to respond appropriately to a SIGBREAK signal. This signal is tied, by default, to the Ctrl + Break key combination. However, neither of these methods is particularly useful if the JVM is hung up somehow.

For these conditions, the IBM Java service team can supply a small stand-alone utility program that is called *jvmdump.exe*. This program takes a single parameter that is the PID of a process. When run, the programme generates a minidump that you can analyze through WinDbg, or translate into a dump-formatter dump in the usual way. (See Chapter 28, “Using the dump formatter,” on page 225 for details.) The *jvmdump* application is provided as-is. If you would like a copy, e-mail [jvmcookbook@uk.ibm.com](mailto:jvmcookbook@uk.ibm.com).

Alternatively, if you have the Microsoft debugging tools installed, you can use Windbg to generate a minidump. See “Generating a user dump file in a hang condition” on page 144 for more information.

## 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

Native memory leaks are not usually relevant to Java so these are discussed very briefly.

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. Simply 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 normally 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.

From the point of view of 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. On a dump, the programmer sees sets of discrete memory addresses.

## Classifying leaks

The following scenarios are possible :

- Windows memory usage is increasing, Java heap is static:
  - Memory leak in application.
  - Memory leak in JNI.
  - Leak with hybrid Java and native objects (very rare).
- Windows memory usage increases because the heap keeps increasing:
  - Memory leak in application Java code. (See "Common causes of perceived leaks" on page 247 below.)
  - Memory leak internal to JVM.

## Tracing leaks

### -Xrunjnichk option

You can use the `-Xrunjnichk` option to trace JNI calls that are made by your JNI code or by any JVM components that use JNI. This helps you to identify incorrect uses of JNI libraries from native code, and can help you to diagnose JNI memory

leaks. Note that **-Xrunjnichk** is equivalent to **-Xcheck:jni**. See “Debugging the JNI” on page 72 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 Appendix D, “Command-line options,” on page 361 for more information.

Some useful techniques are built into the JVM:

- The **-verbose:gc** option
- HeapDump: See Chapter 25, “Using Heapdump,” on page 213
- HPROF tools

## **Using HeapDump to debug memory leaks**

For details about analyzing the Java Heap, see Chapter 25, “Using Heapdump,” on page 213.

---

## **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)

## **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. It 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. This 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. See “How to do heap sizing” on page 20.

### 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 compilation overhead. 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, “Understanding the JIT,” on page 35 and Chapter 29, “JIT problem determination,” on page 243.

### 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 35, “Using the HPROF Profiler,” on page 319. `-Xrunhprof:help` gives you a list of suboptions that you can use with hprof.

---

## Collecting data from a fault condition in 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.
- Heapsdumps. 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 145. This dump is the key to most problems and you collect it by running jextract against the system dump and obtaining a compressed *dumpfile.jar*.
- WebSphere Application Server logs, if you are working in a WebSphere Application Server environment.
- Other data, as determined by your particular problem.

---

# Chapter 17. z/OS problem determination

This chapter describes problem determination on z/OS in:

- “Setting up and checking your z/OS environment”
- “General debugging techniques” on page 152
- “Diagnosing crashes” on page 153
- “Debugging hangs” on page 160
- “Debugging memory leaks” on page 161
- “Debugging performance problems” on page 163
- “Collecting data from a fault condition in z/OS” on page 164

---

## Setting up and checking your z/OS environment

### Maintenance

The Java for z/OS Web site at:

<http://www-1.ibm.com/servers/eserver/zseries/software/java/>

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.

### LE settings

Language Environment (LE) Runtime Options (RTOs) affect operation of C and C++ programs such as the JVM. In general, the options that developers set by using C `#pragma` statements in the code should 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.

### 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 “Debugging memory leaks” 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.

## **z/OS - setting up and checking the environment**

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, by default, generates a Javadump and System Transaction Dump (SYSTDUMP) when any of the following occurs:

- A SIGQUIT signal is received
- The JVM aborts because of a fatal error
- An unexpected native exception occurs (for example, a SIGSEGV, SIGILL, or SIGFPE signal is received)

You can use the **-Xdump** option or the environment variable **JAVA\_DUMP\_OPTS** to change the dumps that are produced on the various types of signal. You can use **JAVA\_DUMP\_TDUMP\_PATTERN** to change the naming convention to which the SYSTDUMP is written as an MVS dataset. For further details, see Chapter 26, "Using core (system) dumps," on page 217.

---

## **General debugging techniques**

### **Starting Javadumps in z/OS**

See Chapter 24, "Using Javadump," on page 203.

### **Starting Heapdumps in z/OS**

See Chapter 25, "Using Heapdump," on page 213.

### **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 formats out all the C-stacks (DSAs) for threads in the process that is the default ASID for the dump.

**ip setd asid(x'007d')**

This command is to set the default ASID use command setdef; 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 ledatal cee,asid(007d),tcb(ttttt)**

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 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 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*).

## Interpreting error message IDs

While working in the OMVS, if you get an error message and if you want to understand exactly what the error message means, go to: <http://www-1.ibm.com/servers/s390/os390/bkserv/lookat/lookat.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 zeroes may 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 or 11 or SIGILL for PIC1).

## Documents to gather

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), an LE dump (CEEDUMP), a JVM trace snap dump, and

## **z/OS - diagnosing crashes**

a formatted JVM dump (javacore). 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) (0x18000000 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=00000007E
| 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 - Please Wait.
IEATDUMP success for DSN='CHAMBER.JVM.TDUMP.CHAMBER1.D060309.T144842'
CEEDUMP in progress - Please Wait.
CEEDUMP success for FILE='/u/chamber/test/ras/CEEDUMP.20060309.144858.196780'
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 dataset to which the transaction dump is written. These locations are configurable and are described in Chapter 22, “Overview of the available diagnostics,” on page 189 and Chapter 23, “Using dump agents,” on page 195.

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 third part native libraries (for example native JDBC drivers).

## **Determining the failing function**

The most practical way to find where the exception occurred is to review either the CEDUMP or the Javadump. Both of these 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 formatter (see Chapter 28, “Using the dump formatter,” on page 225), the dbx debugger, or the IPCS LEDATA VERB Exit.

The CEDUMP 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 DSAs, because this 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 CEDUMP shows where a failure occurred:

## Traceback:

DSA	Entry	E	Offset	Load Mod	Program Unit	Service	Status
00000001	<u>cdump</u>	+00000000		CEQLIB		HLE7709	Call
00000002	@@WRAP@MULTHD		+00000266	CEQLIB			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	_zero	+00000BC4	CEQLIB			HLE7709	Call
00000009	_zero	+0000016E	CEQLIB			HLE7709	Call
0000000A	CEEHDSP	+00003A2C	CEQLIB	CEEHDSP		HLE7709	Call
0000000B	CEEOSIGJ	+00000956	CEQLIB	CEEOSIGJ		HLE7709	Call
0000000C	CELQHROD	+00000256	CEQLIB	CELQHROD		HLE7709	Call
0000000D	CEEOSIGG	-08B3FBBC	CEQLIB	CEEOSIGG		HLE7709	Call
0000000E	CELQHROD	+00000256	CEQLIB	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	CEQLIB	CELQINIT		HLE7709	Call
DSA	DSA Addr	E	Addr	PU Addr	PU Offset	Comp Date	Attributes
00000001	00000001082F78E0	000000001110EB38		0000000000000000	*****	20040312	XPLINK EBCDIC POSIX IEEE
00000002	00000001082F7A20	00000000110AF458		0000000000000000	*****	20040312	XPLINK EBCDIC POSIX Floating Point
00000003	00000001082F7C00	0000000011202988		0000000000000000	*****	20040817	XPLINK EBCDIC POSIX IEEE
00000004	00000001082F8100	0000000011213770		0000000000000000	*****	20040817	XPLINK EBCDIC POSIX IEEE
00000005	00000001082F8200	0000000011219760		0000000000000000	*****	20040817	XPLINK EBCDIC POSIX IEEE
00000006	00000001082F8540	000000001CD4BDA8		0000000000000000	*****	20040817	XPLINK EBCDIC POSIX IEEE
00000007	00000001082F9380	00000000111FF190		0000000000000000	*****	20040817	XPLINK EBCDIC POSIX IEEE
00000008	00000001082F9480	00000000111121E0		0000000000000000	*****	20040312	XPLINK EBCDIC POSIX IEEE
00000009	00000001082FA0C0	0000000011112048		0000000000000000	*****	20040312	XPLINK EBCDIC POSIX IEEE
0000000A	00000001082FA1C0	0000000010DB8EA0		0000000010DB8EA0	00003A2C	20040312	XPLINK EBCDIC POSIX Floating Point
0000000B	00000001082FCAE0	0000000010E3D530		0000000010E3D530	00000956	20040312	XPLINK EBCDIC POSIX Floating Point
0000000C	00000001082FD4E0	0000000010D76778		0000000010D76778	00000256	20040312	XPLINK EBCDIC POSIX Floating Point
0000000D	00000001082FD720	0000000010E36C08		0000000010E36C08	0883FB00	20040312	XPLINK EBCDIC POSIX Floating Point
0000000E	00000001082FE540	0000000010D76778		0000000010D76778	00000256	20040312	XPLINK EBCDIC POSIX Floating Point
0000000F	00000001082FE780	00000000122C66B0		0000000000000000	*****	20040802	XPLINK EBCDIC POSIX IEEE
00000010	00000001082FE880	000000007CD28030		0000000000000000	*****	^C"22^04^FF^FDu^58	XPLINK EBCDIC POSIX IEEE
00000011	00000001082FC80	000000007CD515B8		0000000000000000	*****	20040817	XPLINK EBCDIC POSIX IEEE
00000012	00000001082FD80	00000000111FF948		0000000000000000	*****	20040817	XPLINK EBCDIC POSIX IEEE
00000013	00000001082FE80	000000007CD531A8		0000000000000000	*****	20040817	XPLINK EBCDIC POSIX IEEE
00000014	00000001082FEF80	000000007CD4F148		0000000000000000	*****	20040817	XPLINK EBCDIC POSIX IEEE

## Notes:

1. The stack frame that has a status value of Exception indicates where the crash occurred. In this example, the crash occurs in the function `Java_dumpTest_runTest`.
2. The value under Service for each DSA is the service string. The string is built in the format of `jyymmdd`, where `j` is the identifier for the code owner and `yymmdd` is the build date. A service string like this indicates that the function is part of the JVM. All functions should 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 IBM Service does not recommended you to do that.

A TDUMP can contain multiple Address Spaces. It is important to work with the correct address space associated with the failing java process.

## **Adding the dump file to the IPCS inventory**

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**), sub option 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 (untereded) data set to the IPCS inventory.
3. You may 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**

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
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 ledatal 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 vsmdatal 'summary noglobal'**

Provides a summary of the local data area:

LOCAL STORAGE MAP

Extended LSQA/SWA/229/230	80000000 <- Top of Ext. Private
(Free Extended Storage)	80000000 <- Max Ext. User Region Address 7F4AE000 <- ELSQA Bottom
Extended User Region	127FE000 <- Ext. User Region Top
:	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

## **z/OS - diagnosing crashes**

Input Specifications:

```
Region Requested      =>      3600000
IEFUSI/SMF Specification => SMFL : FFFFFFFF  SMFEL: FFFFFFFF
                                SMFR : FFFFFFFF  SMFER: FFFFFFFF
Actual Limit           => LIMIT:  9FA000  ELIM : 7F606000
```

Summary of Key Information from LDA (Local Data Area) :

```
STRTA = 6000 (ADDRESS of start of private storage area)
SIZA = 9FA000 (SIZE of private storage area)
CRGTP = 7000 (ADDRESS of current top of user region)
LIMIT = 9FA000 (Maximum SIZE of user region)
LOAL = 1000 (TOTAL bytes allocated to user region)
HIAL = 43000 (TOTAL bytes allocated to LSQA/SWA/229/230 region)
SMFL = FFFFFFFF (IEFUSI specification of LIMIT)
SMFR = FFFFFFFF (IEFUSI specification of VVRG)

ESTRA = 10D00000 (ADDRESS of start of extended private storage area)
ESIZA = 6F300000 (SIZE of extended private storage area)
ERGTP = 127FE000 (ADDRESS of current top of extended user region)
ELIM = 7F606000 (Maximum SIZE of extended user region)
ELOAL = 1AFD000 (TOTAL bytes allocated to extended user region)
EHIAL = B36000 (TOTAL bytes allocated to extended LSQA/SWA/229/230)
SMFEL = FFFFFFFF (IEFUSI specification of ELIM)
SMFER = FFFFFFFF (IEFUSI specification of EVVRG)
```

**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_ICL: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 within 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 normally initially thought to be 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	tttyp0003	2:34	java -classpath .Test2Frame

If the value of TIME increases in a few minutes, the process is still using CPU, and is not deadlocked.

For an explanation of deadlocks and how the Javadump tool is used to diagnose them, see “Locks, monitors, and deadlocks (LOCKS)” on page 207.

### 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 as follows:

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)
```

When the console dump has been generated, you can view the Systrace in IPCS to identify threads that are looping. You can do this in IPCS as follows:

```
ip systrace asid(x'007d') time(gmt)
```

This command formats out the system trace entries for all threads that are in address space 0x7d. The **time(gmt)** option converts the TOD clock entries, which are in the system trace, to a human readable form.

From the output produced, you can determine which are the looping threads by identifying patterns of repeated CLCK and EXT1005 interrupt trace entries, and subsequent redispatch DSP entries. You can identify the instruction address range of the loop from the PSWs (Program Status Words) that are traced in these entries.

## The process is performing badly

If you have no evidence of a deadlock or an infinite loop, it is likely that the process is suffering from very bad performance. This 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 actually a hang and should be handled as a performance problem. See “Debugging performance problems” on page 163 for more information.

## Debugging memory leaks

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 third party 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, the Java heap is allocated at startup, along with any subsequent calls to malloc(). A leak of Java objects, therefore, does not cause VSM to issue an abend878 rc10 because of lack of private storage. This abend can be caused only by unbounded growth of storage that is allocated through malloc() for underlying JVM resources requested by JVM components such as AWT or the JIT, or by calls to malloc() from application JNI code and third party native libraries.

## **z/OS - debugging memory leaks**

If you change the LE HEAP setting, you are asking LE to GETMAIN different amounts of initial or incremental storage for use by all C applications. This has no effect on a Java application throwing an OutOfMemoryError. If errors are received because of lack of private storage, you must ensure that the region size is big enough to allocate for the Java heap and for the underlying JVM resources. Note that for TSO/E address spaces, the REGION size for USS processes that are like the JVM inherit from the TSO/E address space, whereas in the case of rlogin or telnet sessions, the region size is determined by the BPXPRMxx parameter **MAXASSIZE**.

## **OutOfMemoryErrors**

The JVM throws a java.lang.OutOfMemoryError (OOM) when the heap is full, and it 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 retried. If successful, the mutator threads resume where they stopped. If the allocation request cannot be fulfilled, an OutOfMemory exception is thrown.

The Garbage Collector uses a mark and sweep algorithm. That is, the Garbage Collector marks every object that is referenced from the stack of a thread, and every object that is referenced from a marked object. Any object on the heap that remains unmarked is cleared up during the sweep phase because it is no longer live.

An OutOfMemory exception occurs when the live object population requires more space than is available in the Java managed heap. It is possible that this can occur not because of an object leak, but because the Java heap is not large enough for the application that is being run. In this case, you can use the **-Xmx** option on the JVM invocation to increase the heap size and remove the problem, as follows:

```
java -Xmx320m MyApplication
```

If the failure is occurring under javac, remember that the compiler is a Java program itself. To pass parameters to the JVM that is created for the compile, use the **-J** option to pass the parameters that you would normally pass directly. For example, the following 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, but increases the time for a failure to occur.

OutOfMemory errors are also generated when a JVM call to malloc() fails. This should normally have an associated error code.

Should an OutOfMemoryError be generated, and no error message is produced, it is assumed that this is because of Java heap exhaustion. At this point, increase the maximum Java heap size to allow for the possibility that the heap is not big enough for the application that is running. Also enable the z/OS heaptump, and 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 will be seen to decrease over time. See “Basic diagnostics (-verbose:gc)” on page 248.

These actions are listed in order of severity. As the number increases, the Garbage Collector is becoming more desperate for memory. A high action number is a good indication of a significant shortage of Java heap space.

A Java object leak is caused when an application retains references to objects that are no longer in use. In a C application, a developer is required to free memory when it is no longer required. A Java developer is required to remove references to objects that are no longer required. The developer normally does this by setting references to null. When this does not happen, the object, and anything that object references in turn, continues to reside on the Java heap and cannot be removed. This typically occurs when data collections are not managed correctly; that is, the mechanism to remove objects from the collection is either not used, or used incorrectly.

---

## 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. See “How to do heap sizing” on page 20.

## **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 compilation overhead. 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, "Understanding the JIT," on page 35 and Chapter 29, "JIT problem determination," on page 243.

## **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 35, "Using the HPROF Profiler," on page 319. **-Xrunhprof:help** gives you a list of suboptions that you can use with hprof.

---

## **Collecting data from a fault condition in z/OS**

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 formatter (see Chapter 28, "Using the dump formatter," on page 225), the dbx debugger, or IPCS (Interactive Problem Control System).
- CEDUMP - 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 Chapter 27, "Using method trace," on page 221 and Chapter 33, "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).
- Trace data from other products or components (for example LE traces or the Component trace for z/OS UNIX).
- Heapdumps, if generated automatically, are essential. They are also essential if you have a memory or performance problem.

---

## Chapter 18. 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 book 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 79 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](http://java.sun.com/j2se/1.5/pdf/jdk50_ts_guide.pdf).



---

## Chapter 19. 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 book 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 79 and submit a bug report.



---

## Chapter 20. 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 execute 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, it is likely that if a problem occurs, both sides will record an exception or unusual behavior.

This chapter 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 171
- “ORB exceptions” on page 172
- “Interpreting the stack trace” on page 174
- “Interpreting ORB traces” on page 175
- “Common problems” on page 178
- “IBM ORB service: collecting data” on page 180

---

### Identifying an ORB problem

When you find a problem that you think is related to CORBA or RMI, a knowledge of the constituents of the IBM ORB component can be very helpful.

### What the ORB component contains

The ORB component contains the following:

- IBM Java ORB 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 problem

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.

### Packaging

Table 5. Packaging

	IBM platforms
Runtime classes	jre/lib/ibmorb.jar
Tools classes	lib/tools.jar
CORBA API classes	jre/lib/ibmorbapi.jar
Runtime support	jre/lib/ibmcfw.jar
rmic wrapper	None
idlj wrapper	None
tnameserv wrapper	None
orbd wrapper	None

### ORB versions

The ORB component carries a few version properties that you can display by invoking 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)

**Note:** Items 2 and 3 are alternative methods for reaching the same class.

## Limitation with bidirectional GIOP

Bidirectional GIOP is not supported.

## Debug properties

**Attention:** Do not turn on 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 output file is generated, 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 field, the file name defaults to the format `orbtrc.DDMMMYYYY.HHMM.SS.txt`, where D=Day; M=Month; Y=Year; H=Hour (24 hour format); m=Minutes; S=Seconds. Note that 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. Every incoming and outgoing GIOP message will be output to the trace log. You can set this property independently from Debug; this is useful if you want to look only at the flow of information, and you are not too worried about 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 example:

```
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

You are using this chapter because you think that your problem is related to the ORB. Unless your application is doing nothing or giving you the wrong result, it is likely that your log file or terminal is 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 is also true 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 invoked 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 invoked.
- **BAD\_PARAM:** This exception is thrown when a parameter that is passed to a call is out of range or otherwise considered illegal. 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 invoked 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. (This can happen 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.)

## Completion status and minor codes

Each system exception has two pieces of data that are associated with it:

- 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 this 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 (see the section below) 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 337 gives diagnostic information for the most-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. Here is a selection of affected methods:

*Table 6. 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

*Table 6. Methods affected when running with Java 2 SecurityManager (continued)*

Class/Interface	Method	Required permission
org.omg.CORBA.portable.ObjectImpl	_is_a	java.net.SocketPermission connect
org.omg.CORBA.portable.ObjectImpl	_non_existent	java.net.SocketPermission connect
org.omg.CORBA.portable.ObjectImpl	OutputStream _request (String, boolean)	java.net.SocketPermission connect
org.omg.CORBA.portable.ObjectImpl	_get_interface_def	java.net.SocketPermission connect
org.omg.CORBA.Request	invoke	java.net.SocketPermission connect
org.omg.CORBA.Request	send_deferred	java.net.SocketPermission connect
org.omg.CORBA.Request	send_oneway	java.net.SocketPermission connect
javax.rmi.PortableRemoteObject	narrow	java.net.SocketPermission connect

If your program uses any of these methods, ensure that it is granted the necessary permissions.

## Interpreting the stack trace

Whether the ORB is part of a middleware application or you are using a Java standalone 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.io.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.io.CDRInputStream.read_value(CDRInputStream.java:1429)
at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie._invoke(_EJSRemoteStatelessPmiService_Tie.java:613)
at com.ibm.CORBA.iiop.ExtendedServerDelegate.dispatch(ExtendedServerDelegate.java:515)
at com.ibm.CORBA.iiop.ORB.process(ORB.java:2377)
at com.ibm.CORBA.iiop.OrbWorker.run(OrbWorker.java:186)
at com.ibm.ejs.oa.pool.ThreadPool$PooledWorker.run(ThreadPool.java:104)
at com.ibm.ws.util.CachedThread.run(ThreadPool.java:137)
```

### Description string

The example stack trace shows that the application has caught a CORBA org.omg.CORBA.MARSHAL system exception. After the MARSHAL exception, some extra information is provided in the form of a string. This string should specify minor code, completion status, and other information that is related to the problem. Because CORBA system exceptions are alarm bells for an unusual condition, they also hide inside what the real exception was.

Usually, the type of the exception is written in the message string of the CORBA exception. The trace shows that the application was reading a value (`read_value()`) when an `IllegalAccessException` occurred that was associated to class `com.ibm.ws.pmi.server.DataDescriptor`. This is a hint of the real problem and should be investigated first.

## Nested exceptions

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.

## Interpreting ORB traces

The ORB trace file contains messages, trace points, and wire tracing. This section describes the various types of trace.

### 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 invoked. 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 (88 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 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:

## ORB - interpreting ORB traces

```
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 invoked. 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

Here is an example of comm (wire) 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 0000006C 00010200
0000000D 392E3230 2E313738 2E313336
00001324 0000001A 4C4D4249 00000010
15074A96 00100000 00080000 00000000
00000000 00000002 00000001 00000018
00000000 00010001 00000001 00010020
00010100 00000000 49424D0A 00000008
00000000 14000005

Data Offset: 11c
// raw data that goes in the wire in numbered rows of 16 bytes and the corresponding ASCII
decoding
0000: 47494F50 01020000 00000114 00000005 GIOP.....
0010: 03000000 00000000 0000001A 4C4D4249 .....LMBI
0020: 00000010 14F94CA4 00100000 00080000 .....L.....
```

```

0030: 00000000 00000000 00000008 6D657373 .....mess
0040: 61676500 00000003 49424D12 00000008 age.....IBM.....
0050: 00000000 14000005 00000001 0000000C .....
0060: 00000000 00010001 00010100 00000006 .....
0070: 000000A8 00000000 00000028 49444C3A .....(IDL:
0080: 6F6D672E 6F72672F 53656E64 696E6743 omg.org/SendingC
0090: 6F6E7465 78742F43 6F646542 6173653A ontext/CodeBase:
00A0: 312E3000 00000001 00000000 0000006C 1.0.....1
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.

## 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 in the machine 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-iop 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

## ORB - interpreting ORB traces

for purposes that can be vendor-specific (such as the IBM Partner version that is described in the IOR in Chapter 6, "Understanding the ORB," on page 39).

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 communicate normally through ASCII. 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 this happens, the most likely condition (and most difficult to solve) is a deadlock of threads. In this condition, it is important to know whether the machine that on which you are running has more than one CPU.

A simple test that you can do is to keep only one CPU running 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 milliseconds.

Also, set the RequestTimeout to a value other than 0. Because a reply to a request is often large, allow more time; for example, 10000 milliseconds. These values are suggestions and might be too low for slow connections. When a request times out, 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 across 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 time-out, and problems of waiting threads might occur.

If the problem appears to be a deadlock or hang, capture the Javadump information. Do this once, then wait for a minute or so, and do it again. A comparison of the two snapshots shows whether any threads have changed state. For information about how to do this operation, see “Triggering a Javadump” on page 204.

In general, stop the application, enable the orb traces (see previous section) 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 invoked**

This operation outputs:

```
(org.omg.CORBA.COMM_FAILURE)
Hello Client exception:
    org.omg.CORBA.COMM_FAILURE:minor code:1 completed:No
        at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:145)
        at com.ibm.rmi.iiop.ConnectionTable.get(ConnectionTable.java:77)
        at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)
        at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:75)
        at com.ibm.rmi.corba.ClientDelegate.createRequest(ClientDelegate.java:440)
        at com.ibm.rmi.corba.ClientDelegate.is_a(ClientDelegate.java:571)
        at org.omg.CORBA.portable.ObjectImpl._is_a(ObjectImpl.java:74)
        at org.omg.CosNaming.NamingContextHelper.narrow(NamingContextHelper.java:58)
        com.sun.jndi.cosnaming.CNCTx.callResolve(CNCTx.java:327)
```

## **Client and server are running, but not naming service**

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)
```

## ORB application hangs

```
at com.ibm.rmi.corba.InitialNamingClient.cachedInitialReferences(InitialNamingClient.j  
at com.ibm.rmi.corba.InitialNamingClient.resolve_initial_references(InitialNamingClien  
at com.ibm.rmi.corba.ORB.resolve_initial_references(ORB.java:1269)  
.....
```

You must start the Java IDL name server before an application or applet starts that uses its naming service. Installation of the Java IDL product creates a script (Solaris: tnameserv) or executable file that starts the Java IDL name server.

Start the name server so that it runs in the background. If you do not specify otherwise, the name server listens on port 2809 for the bootstrap protocol that is used to implement the ORB resolve\_initial\_references() and list\_initial\_references() methods.

Specify a different port, for example, 1050, as follows:

```
tnameserv -ORBInitialPort 1050
```

Clients of the name server must be made aware of the new port number. Do this by setting the org.omg.CORBA.ORBInitialPort property to the new port number when you create the ORB object.

## Running the client with MACHINE2 (client) unplugged from the network

Your output is:

```
(org.omg.CORBA.TRANSIENT CONNECT_FAILURE)
```

```
Hello Client exception:Problem contacting address:corbaloc:iiop:machine2:2809/NameService  
javax.naming.CommunicationException:Problem contacting address:corbaloc:iiop:machine2:2809/N  
is org.omg.CORBA.TRANSIENT:CONNECT_FAILURE (1)minor code:4942F301 completed:No  
at com.ibm.CORBA.transport.TransportConnectionBase.connect(TransportConnectionBase.jav  
at com.ibm.rmi.transport.TCPTTransport.getConnection(TCPTTransport.java:178)  
at com.ibm.rmi.iiop.TransportManager.get(TransportManager.java:79)  
at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:131)  
at com.ibm.rmi.iiop.GIOPImpl.createRequest(GIOPImpl.java:98)  
at com.ibm.CORBA.iiop.ClientDelegate._createRequest(ClientDelegate.java:2096)  
at com.ibm.CORBA.iiop.ClientDelegate.createRequest(ClientDelegate.java:1264)  
at com.ibm.CORBA.iiop.ClientDelegate.createRequest(ClientDelegate.java:1177)  
at com.ibm.rmi.corba.InitialNamingClient.resolve(InitialNamingClient.java:252)  
at com.ibm.rmi.corba.InitialNamingClient.cachedInitialReferences(InitialNamingClient.j  
at com.ibm.rmi.corba.InitialNamingClient.resolve_initial_references(InitialNamingClien  
at com.ibm.rmi.corba.InitialReferenceClient.resolve_initial_references(InitialReferenc  
at com.ibm.rmi.corba.ORB.resolve_initial_references(ORB.java:3211)  
at com.ibm.rmi.iiop.ORB.resolve_initial_references(ORB.java:523)  
at com.ibm.CORBA.iiop.ORB.resolve_initial_references(ORB.java:2898)  
.....
```

---

## IBM ORB service: collecting data

This section describes how to collect data about ORB problems.

### 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 (not only on customers' machines, but on a similar setup configuration).
- The JIT is disabled (see Chapter 29, "JIT problem determination," on page 243).

Also:

1. Disable additional CPUs.
2. 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. Remember that even with 1 GB of physical RAM, Java can use only 512 MB independently of what **-Xmx** is set to.
3. Check physical network problems (firewalls, com links, routers, DNS name servers, and so on). These are the major causes of CORBA COMM\_FAILURE exceptions. As a test, ping your own machine name.
4. 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.

## Data to be collected

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 chapter).
- If the problem is in `rmic -iiop` or `rmic -idl`, set the options: `-J-Djavac.dump.stack=1 -Xtrace`, and capture the output.
- Normally 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.

## **ORB - IBM ORB service: collecting data**

---

## Chapter 21. NLS problem determination

The JVM contains built-in support for different locales. This chapter provides an overview of locales, with the main focus on fonts and font management.

- “Overview of fonts”
- “Font utilities” on page 184
- “Common problems and possible causes” on page 184

---

### Overview of fonts

When you want to display text, either in SDK components (AWT or Swing), on the console or in any application, characters have to 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 may be represented by more than one glyph (for example, in Indic scripts).

A font is a set of glyphs, where 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

A 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 be displayed in various faces. For example: Normal, Italic, and Oblique

#### Font variant

This property determines whether the font should be 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

This refers to the boldness or the lightness of the glyph to be used.

#### Font size

This property 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 it should be), `xset -q` output also shows the fonts that are bundled with the Developer Kit.

## NLS - overview of fonts

Use xset +fp and xset -fp to add and remove the font path respectively.

### 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.

---

## Font utilities

### Font utilities on AIX, Linux, and z/OS

#### xfd

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 problems and possible causes

### 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.

### Character displayed in the console but not in the SDK Components and vice versa.

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 `x1fd <physical font name>` command to check whether the system can recognize the character or not.

### Character not displayed in TextArea or TextField

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/PushB.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
```

Note that the Motif library is statically linked into the Linux JVMs, so it is not possible to use this technique there.

## **NLS - common problems and possible causes**

---

## Part 4. Using diagnostic tools

This part of the book describes how to use the diagnostic tools that are available. The chapters are:

- Chapter 22, “Overview of the available diagnostics,” on page 189
- Chapter 23, “Using dump agents,” on page 195
- Chapter 24, “Using Javadump,” on page 203
- Chapter 25, “Using Heapdump,” on page 213
- Chapter 26, “Using core (system) dumps,” on page 217
- Chapter 27, “Using method trace,” on page 221
- Chapter 28, “Using the dump formatter,” on page 225
- Chapter 29, “JIT problem determination,” on page 243
- Chapter 30, “Garbage Collector diagnostics,” on page 247
- Chapter 31, “Class-loader diagnostics,” on page 265
- Chapter 32, “Shared classes diagnostics,” on page 267
- Chapter 33, “Tracing Java applications and the JVM,” on page 283
- Chapter 34, “Using the Reliability, Availability, and Serviceability Interface,” on page 305
- Chapter 35, “Using the HPROF Profiler,” on page 319
- Chapter 36, “Using the JVMTI,” on page 325
- Chapter 37, “Using DTFJ,” on page 327

**Notes:**

1. JVMMI is not supported on the Version 5 platforms described in this book.
2. JVMPPI is now a deprecated interface, replaced by the HPROF Profiler.



---

## Chapter 22. Overview of the available diagnostics

This chapter summarizes the diagnostic information that can be produced by the JVM and points towards other chapters that give more details on its use in problem determination. It also describes the range of supplied tools that can be used to post-process this information and help with problem determination. Subsequent chapters in this part of the book 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

Problems fall into four categories:

1. Crashes
2. Hangs
3. Memory leaks
4. Poor performance

During problem determination, one of the first objectives is to identify the most probable area where the problem originates. Many problems that appear to be a Java problem originate elsewhere. Areas where problems can arise include:

- The JVM itself
- Native code
- A badly written or overstretched Java application
- A system or system resource
- A subsystem (such as database code)
- Hardware

Different tools and different diagnostic information might be needed 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. Many other tools are supplied by hardware or system software vendors (such as system debuggers). They are not described in this book but might be useful in problem determination.

---

### Platforms

IBM provides and supports Java on a number of platforms. These platforms can be divided into groups:

1. Linux
2. Windows
3. AIX (Power PC)
4. z/OS (previously called S/390)

The majority of tools discussed in this book 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. (For instance, system debuggers tend to be tied to a particular type of system.)

---

## Summary of diagnostic information

A running IBM JVM has inbuilt 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 26, “Using core (system) dumps,” on page 217). 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 be discussed here. Appendix B, “Environment variables,” on page 339 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 or when the user enters a reserved key combination (for example, **Ctrl-Break** on Windows). 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 24, “Using Javadump,” on page 203 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 within 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 Heaproofs. See Chapter 25, “Using Heapdump,” on page 213 for more details.

### System dumps

By default, the JVM generates system dumps, which 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 use the **-Xdump:system** option to produce system dumps when other events occur.

### 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 output 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 preformatted. For more information, see Chapter 33, “Tracing Java applications and the JVM,” on page 283.

#### 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 very similar to normal trace and requires the use of a supplied trace formatter so that you can read it. See Chapter 33, “Tracing Java applications and the JVM,” on page 283 for more details on the contents and control of snap traces.

#### 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 (more usually) problems in the design of user applications. Numerous other options affect the nature and amount of Garbage Collector diagnostic information produced. See Chapter 30, “Garbage Collector diagnostics,” on page 247 for more information.

#### Other data

Special options are available for producing diagnostic information relating to

- The JIT (see Chapter 29, “JIT problem determination,” on page 243)
- Class loading (see Chapter 31, “Class-loader diagnostics,” on page 265)
- Shared classes (see Chapter 32, “Shared classes diagnostics,” on page 267)

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 35, “Using the HPROF Profiler,” on page 319 for more details.

## Summary of cross-platform tooling

IBM has several cross-platform diagnostic tools. They apply to the different types of problem described above. 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 25, “Using Heapdump,” on page 213 for more information.

### Cross-platform dump formatter

The cross-system dump formatter is a more advanced tool than Javadump. It 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 code to extract and package (compress) data from the dump generated by the native operating system
2. jdmpview - a Java tool to analyze that data

## diagnostics - cross-platform tools

The formatter "understands" the JVM and can be used to analyze its internals. It is a useful tool to debug unexpected terminations of the JVM. You must understand the JVM internals and infrastructure to use this tool. 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 machine 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 28, "Using the dump formatter," on page 225.

## JVMTI tools

The JVMTI (JVM Tool Interface) is the latest programming interface for use by tools. It replaces the Java Virtual Machine Profiler Interface (JVMPPI) and the Java Virtual Machine Debug Interface (JVMDI) and, although JVMPPI and JVMDI are currently still available (but deprecated), the intention is to remove them at some future date. For information on the JVMTI, see Chapter 36, "Using the JVMTI," on page 325. The HPROF tool provided with the SDK has been updated to use the JVMTI; see Chapter 35, "Using the HPROF Profiler," on page 319.

## JVMPPI tools

**The JVMPPI is now a deprecated interface..**

JVMPPI 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 36, "Using the JVMTI," on page 325. 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 JVMPPI specification. Existing JVMPPI-based tools (such as the vendor tools JProbe, OptimizeIt, TrueTime, and Quantify) that use the JVMPPI should continue to work. The IBM SDK provided tool HPROF has been updated to use the JVMTI; see Chapter 35, "Using the HPROF Profiler," on page 319).

## JPDA tools

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, these tools are best suited to tracing leaks or the conditions before a termination or hang, if these conditions are repeatable.

An example of such a tool is the debugger that is bundled with Eclipse for Java.

## 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 means that you can tailor trace to be a relatively minor hit on performance.

The IBM Virtual Machine for Java contains a large amount of embedded trace. **In this release, maximal tracing is switched on by default for a small number of 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, application programmers or diagnosticians can place their own application trace points in their class code and also activate tracing for entry and exit against all methods in all classes or for a selection of methods in a selection of classes. Application and method traces will be interleaved in the trace buffers with the JVM embedded trace points, allowing detailed analysis of the routes taken through the code.

Trace applies mainly to performance and leak problem determination, although trace data might 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 Chapter 27, “Using method trace,” on page 221 and Chapter 33, “Tracing Java applications and the JVM,” on page 283 for more details. You have to make a considerable effort to master the workings of trace. However, it is an extremely 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 34, “Using the Reliability, Availability, and Serviceability Interface,” on page 305 for details. You need some programming skills and tools to be able to use this interface.



---

## Chapter 23. Using dump agents

Dump agents are set up during JVM initialization. They enable you to use events occurring within the JVM, such as Garbage Collection, thread start, or JVM termination, to initiate one of four types of dump or to launch an external tool. Default dump agents are set up at JVM initialization. They are sufficient for most cases, but the use of the **-Xdump** option on the command line allows more detailed configuration of dump agents. The total set of options and sub-options available under **-Xdump** is very flexible and there are many examples presented in this chapter to show this flexibility.

The **-Xdump** option allows you 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 chapter describes:

- “Help options”
- “Dump types and triggering” on page 197
- “Types of dump agents - examples” on page 197
- “Default dump agents” on page 199
- “Default settings for dumps” on page 200
- “Limiting dumps using filters and range keywords” on page 201
- “Removing dump agents” on page 201
- “Controlling dump ordering” on page 201
- “Controlling dump file names” on page 202

---

### Help options

You can obtain help on the various usage aspects of **-Xdump** by using **java -Xdump:help**.

*Table 7. Usage from java -Xdump:help*

Command	Result
<b>-Xdump:help</b>	Print general dump help
<b>-Xdump:none</b>	Ignore all previous and default dump options
<b>-Xdump:events</b>	List available trigger events
<b>-Xdump:request</b>	List additional VM requests
<b>-Xdump:tokens</b>	List recognized label tokens
<b>-Xdump:dynamic</b>	Enable support for pluggable agents
<b>-Xdump:what</b>	Show registered agents on startup
<b>-Xdump:&lt;type&gt;:help</b>	Print detailed dump help
<b>-Xdump:&lt;type&gt;:none</b>	Ignore previous dump options of this type
<b>-Xdump:&lt;type&gt;:defaults</b>	Print and update default settings for this type
<b>-Xdump:&lt;type&gt;</b>	Request this type of dump (using defaults)

## Using dump agents - help options

Table 8. Types of dump

Valid types of dump	Description
-Xdump:console	Basic thread dump to stderr
-Xdump:system	Capture raw process image. See Chapter 26, "Using core (system) dumps," on page 217.
-Xdump:tool	Run command line program
-Xdump:java	Write application summary. See Chapter 24, "Using Javadump," on page 203.
-Xdump:heap	Capture heap graph. See Chapter 25, "Using Heapdump," on page 213.
-Xdump:snap	Take a snap of the trace buffers

As an example:

```
java -Xdump:heap:none -Xdump:heap:events=fullgc class [args...]
```

turns off default Heapdumps and then requests a Heapdump on every full GC.

As can be seen from Table 7 on page 195, further help is available for the assorted suboptions under **-Xdump**. In particular, **java -Xdump:events** shows the available keywords used to specify the events that can be used.

You must filter class events (such as load, throw, and uncaught) by class name. For guidance, see "Limiting dumps using filters and range keywords" on page 201.

Table 9. Keywords

Supported event keywords	Event hook
gpf	ON_GP_FAULT
user	ON_USER_SIGNAL
abort	ON_ABORT_SIGNAL
vmstart	ON_VM_STARTUP
vmstop	ON_VM_SHUTDOWN
load	ON_CLASS_LOAD
unload	ON_CLASS_UNLOAD
throw	ON_EXCEPTION_THROW
catch	ON_EXCEPTION_CATCH
brkpoint	ON_BREAKPOINT
framepop	ON_DEBUG_FRAME_POP
thrstart	ON_THREAD_START
blocked	ON_THREAD_BLOCKED
thrstop	ON_THREAD_END
expand	ON_HEAP_EXPAND
fullgc	ON_GLOBAL_GC
uncaught	ON_EXCEPTION_DESCRIBE
slow	ON_SLOW_EXCLUSIVE_ENTER
any	*

## Dump types and triggering

The main purpose of the **-Xdump** stanza on the command line is to link *events* to a *dump type* (**-Xdump:tool** is a little misleading, because it is a command, not a dump). Thus, **-Xdump:heap:events=vmstop** is an instruction to JVM initialization to create a dump agent that produces a Heapdump whenever the vmstop event happens. The JVM is constructed to generate at the appropriate time the events listed in Chapter 26, “Using core (system) dumps,” on page 217.

You can have multiple **-Xdump** stanzas on the command line and also multiple dump types driven by one or multiple events. Thus, **-Xdump:heap+java:events=vmstart+vmstop** would create a dump agent that would drive both heap and Java dump production when either a vmstart or vmstop event was encountered.

Note that multiple **-Xdump** stanzas on the command line can be used to create multiple agents at JVM initialization; these agents are chained together and all evaluated whenever an event occurs. The dump agent processing ensures that multiple **-Xdump** stanzas are optimized. You can use the **-Xdump:what** stanza to clarify this optimization.

The keyword **events** is used as the prime trigger mechanism. However, there are a number of additional keywords that you can use to further control the dump produced (**request** and **tokens**, for example) or limit its production to a smaller range of circumstances; use **-Xdump<type>:help** to find these.

---

## Types of dump agents - examples

This section presents several examples of the use of **-Xdump**, based around each dump type, to illustrate the style of syntax and the generated function. The examples given are deliberately simplistic to limit the size of the output.

As you can see from using **-Xdump:help**, there are many dump types to consider.

### Console dumps

Console dumps are simple dumps, in which the status of every Java thread is written to stderr. Some output of this type is shown below. Note the use of the **range=1..1** suboption to control the amount of output to just one thread start and stop (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(00035B24)
Status=Running
ThreadName=main(00035A1C)
Status=Waiting
Monitor=00035128 (VM sig quit)
Count=0
Owner=(00000000)
~~~~~ Console dump ~~~~~~
```

### System dumps

System dumps involve dumping a whole frozen 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

## Types of dump agents - examples

of file space and take several minutes to complete. Shown below is an example of invoking a system dump on a Windows 32-bit machine. Note the use of **request=nodumps+exclusive+prepwalk** in this example, to ensure that this dump is not interrupted by other dumps and that the Java heap is walkable, enabling the objects within the heap to be processed under jextract or jdmpview (the equivalent of **-Xdump:heapdump** in the previous release). Note also that the file name is overridden from the default in this example.

```
java -Xdump:system:events=vmstop,request=nodumps+exclusive+prepwalk,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".
```

## Tool option

The **tool** option allows external processes to be spawned when an event occurs. Consider the following simple example, which displays start of pid and end of pid information (note the use of the token %pid). More realistic examples would invoke a debugging tool, and that is the default taken if you use (for example) **-Xdump:tool:events=.....**

```
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".
```

## Javadumps

Java dumps 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 (which also shows the use of the **filter** keyword) in which a Javadump is produced on the loading of a class is shown below.

```
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".
```

## Heapdumps

From Version 1.4.2, Service Refresh 2, Heapdumps now produce phd format files by default (as described in Chapter 25, “Using Heapdump,” on page 213) unless overridden. The example below shows the production of a Heapdump. Note that in this case the normal production of a phd file only has been augmented by the

use of the **opts=** suboption to produce both phd and classic (.txt) heapdumps (equivalent to using the environment variable **IBM\_JAVA\_HEAPDUMP\_TEST**).

```
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".
```

## Snap traces

From Version 5.0, snap traces join the range of outputs controlled by **-Xdump**. The example below shows the production of a snap trace.

```
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".
```

Snap traces are given sequential numbers (Snap0001 then Snap0002). The produced snap traces are encoded and require the use of the trace formatter for further analysis.

## 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 the **JAVA\_DUMP\_OPTS** environment variable and further override the set by the use of **-Xdump** on the command line (See “Removing dump agents” on page 201).

The **-Xdump:what** option on the command line is very useful for determining which dump agents exist at the completion of startup and can help resolve issues about what has overridden what. Below is sample output showing the default dump agents that are in place when there have been no overrides by using environment variables.

```
java -Xdump:what

Registered dump agents
-----
dumpFn=doSystemDump
events=gpf+abort
```

## Default dump agents

```
filter=
label=C:\sdk\jre\bin\core.%Y%m%d.%H%M%S.%pid.dmp
range=1..0
priority=999
request=serial
opts=
-----
dumpFn=doSnapDump
events=gpf+abort
filter=
label=C:\sdk\jre\bin\Snap%seq.%Y%m%d.%H%M%S.%pid.trc
range=1..0
priority=500
request=serial
opts=
-----
dumpFn=doSnapDump
events=uncaught
filter=java/lang/OutOfMemoryError
label=C:\sdk\jre\bin\Snap%seq.%Y%m%d.%H%M%S.%pid.trc
range=1..4
priority=500
request=serial
opts=
-----
dumpFn=doHeapDump
events=uncaught
filter=java/lang/OutOfMemoryError
label=C:\sdk\jre\bin\heapdump.%Y%m%d.%H%M%S.%pid.phd
range=1..4
priority=40
request=exclusive+prewalk
opts=PHD
-----
dumpFn=doJavaDump
events=gpf+user+abort
filter=
label=C:\sdk\jre\bin\javacore.%Y%m%d.%H%M%S.%pid.txt
range=1..0
priority=10
request=exclusive
opts=
-----
dumpFn=doJavaDump
events=uncaught
filter=java/lang/OutOfMemoryError
label=C:\sdk\jre\bin\javacore.%Y%m%d.%H%M%S.%pid.txt
range=1..4
priority=10
request=exclusive
opts=
-----
```

---

## Default settings for dumps

To view the default settings for a particular dump type, use:

**-Xdump:<type>:defaults**

You can change these defaults 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 javadump agent; it updates the default settings for  
| dump agents. Further dump agents will then create dump files using this  
| specification for filenames, unless overridden.

## **Limiting dumps using filters and range keywords**

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.

You can filter class events (such as load, throw, and uncaught) by class name:

| **-Xdump:java:events=throw,filter=java/lang/OutOfMem\* # prefix**  
| **-Xdump:java:events=throw,filter=\*MemoryError # suffix**  
| **-Xdump:java:events=throw,filter=\*Memory\* # substring**

| You can filter the JVM shutdown event by using one or more exit codes:

| **-Xdump:java:events=vmstop,filter=#129..192#-42#255**

You can start and stop dump agents on a particular occurrence of a JVM event by using the range suboption:

| **-Xdump:java:events=fullgc,range=100..200**

Note that **range=1..0** against an event means "on every occurrence".

## **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,

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

turns off all heapdumps (including default agents) but leaves javadump enabled.

## **Controlling dump ordering**

In some situations, one event can generate multiple dumps. For example, examination of the output from `java -Xdump:what` shows that a gpf event produces a snap trace, a java dump, and a system dump. The agents that produce each dump run sequentially and their order is determined by the priority keyword set for each agent. In the example, the system dump would run first (priority 999), the snap dump second (priority 500), and the javadump last (priority 10). An example of setting the priority from the command line is:

## Removing dump agents

```
java -Xdump:heap:events=vmstop,priority=123
```

The maximum value allowed for priority is 999 and the higher the priority the higher a dump agent will sit in the chain.

If you do not specifically set a priority then default values are taken based on the dump type. The default priority, as well as the other default values for a particular type of dump, can be displayed by using the defaults option on the **-Xdump** invocation, for example:

```
C:\sdk\jre\bin>java -Xdump:heap:defaults
```

Default -Xdump:heap settings:

```
events=gpf+user
filter=
file=C:\sdk\jre\bin\heapspace.%Y%m%d.%H%M%S.%pid.phd
range=1..0
priority=40
request=exclusive+prewalk
opts=PHD
```

---

## Controlling dump file names

Dumps are created by default in the working directory. Most often, this directory is the one from which the application was launched. If the dump cannot be created there for any reason (such as it being a read-only location), alternative locations are tried.

The pattern in the file setting for that particular dump agent determines the name of the dump. The defaults (use `java -Xdump:<type>:defaults`) are usually sufficient; however, you can use the file keyword on the command line to set your own file name and location.

---

## Chapter 24. Using Javadump

Javadump produces files that contain diagnostic information related to the JVM and a Java application captured at a point during execution. For example, the information can be about the operating system, the application environment, threads, stacks, locks, and memory. The exact contents depend on the platform on which you are running. The files produced by Javadump are called "Javadump files". By default, a Javadump occurs when the JVM terminates unexpectedly. A Javadump can also be triggered by sending specific signals to the JVM. Javadumps are human readable and it is often the prime tool used in diagnosing application failure.

The preferred way to control the production of Javadumps is by enabling dump agents (see Chapter 23, "Using dump agents," on page 195) using **-Xdump:java:** on application startup. You can also control Javadumps by the use of environment variables. See "Environment variables and Javadump" on page 210.

Default agents are in place that (if not overridden) create Javadumps when the JVM terminates unexpectedly or when an OUT\_OF\_MEMORY condition occurs. Javadumps are also triggered by default when specific signals are received by the JVM.

**Note:** Javadump is also known as **Javacore**. This is NOT the same as a **core file** (that is an operating system feature that can be produced by any program, not just the JVM).

This chapter describes:

- "Enabling a Javadump"
- "The location of the generated Javadump"
- "Triggering a Javadump" on page 204
- "Interpreting a Javadump" on page 204
- "Environment variables and Javadump" on page 210

---

### Enabling a Javadump

Javadumps are enabled by default. Javadump production can be turned off using **-Xdump:java:none**. This is not recommended as Javadumps are an essential diagnostics tool.

Use the **-Xdump:java** option to give more fine-grained control over the production of Javadumps. See Chapter 23, "Using dump agents," on page 195 for more information.

---

### The location of the generated Javadump

When a Javadump is triggered, the JVM checks each of the following locations for existence and write-permission, and stores the Javadump in the first one available. You must have enough free disk space (possibly up to 2.5 MB) for the Javadump file to be written correctly.

1. The location specified by the **IBM\_JAVACOREDIR** environment variable if set (**\_CEE\_DMPTARG** on z/OS).

## Location of the generated Javadump

2. The current working directory of the JVM processes.
3. The location specified by the **TMPDIR** environment variable, if set.
4. The /tmp directory (C:\Temp on Windows).
5. If the Javadump cannot be stored in any of the above, it is put to STDERR.

The file name is of the following form: javacore.%Y%m%d.%H%M%S.%pid.txt (where %pid is the process ID).

---

## Triggering a Javadump

A Javadump is triggered when one of the following occurs:

- **A fatal native exception** occurs in the JVM (not a Java Exception).
- **The JVM has insufficient memory to continue operation.** Often caused by heap expansion and compaction.
- **You send a signal** to the JVM from the operating system.
- **You use the JavaDump() method** within Java code that is being executed.

The exact conditions in which you get a Javadump vary depending on whether the default dump agents have been overridden.

A "fatal" exception is one that causes the JVM to terminate. The JVM handles this by producing a system dump followed by a snap trace file and then terminating the process.

In the user-controlled cases (the latter two), the JVM stops execution, performs the dump, and then continues execution.

The signal for Linux and AIX is SIGQUIT. Use the command kill -3 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 **CTRL+V**.

In Windows, the dump is initiated by using **CTRL+Break** in the command window that started Java. To do this in a WebSphere Application Server environment, use the DrAdmin utility.

The class **com.ibm.jvm.Dump** 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 as are described in "Enabling a Javadump" on page 203.

---

## Interpreting a Javadump

This section gives examples of the information contained in a Javadump and how it can be useful in problem solving.

The information in a Javadump file is essentially the same, regardless of the platform on which you are running the JVM. However, certain platforms might provide more information about fatal exceptions.

The content and range of information in a Javadump is **not guaranteed across releases**. In some cases, information might be missing because of the nature of a crash.

## Javadump tags

The Javadump file contains sections separated by eyecatcher title areas to aid readability of the Javadump. The first such eyecatcher is shown below:

```
-----  
0SECTION TITLE subcomponent dump routine  
=====
```

Different sections contain different tags, which make the file easier to parse for performing simple analysis. An example tag (1CIJAVAVERSION) is shown below:

```
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 component that wrote the message. For example, TI, XH, CI, LK, XE. CI in the example above.
- The remainder is a unique string, JAVAVERSION in the example above.
- 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 Z/OS dump. The components are highlighted for clarification:

```
:  
-----  
0SECTION TITLE subcomponent dump routine  
=====  
1TISIGINFO Dump Event "user" (00004000) received  
1TIDATETIME Date: 2005/08/23 at 10:20:50  
1TIFILENAME Javacore filename: /u/test/javacore.20050823.102050.16908401.txt  
-----  
0SECTION GPINFO subcomponent dump routine  
=====  
2XHOSLEVEL OS Level : z/OS 06.00  
2XHCPUS Processors -  
3XHCPUARCH Architecture : s390x  
3XHNUMCPUS How Many : 2
```

For the rest of the chapter, 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. The example below shows some output taken from a simple test program running on Windows XP calling (using JNI) an external function that executes a strcpy into a protected location. strcpy produces an access violation that has an exception code (on Windows) of 0xC0000005. This is commonly referred to as a “general protection fault” or gpf.

The TITLE section of the Javadump shows basic information about the event that caused the generation of the Javadump, the time it was taken, and its name.

The GPINFO section 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, which can vary depending on whether it is Windows, Linux,

## interpreting a Javadump

| z/OS, or AIX, and whether it is a 32- or 64-bit 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 inside MVSCR71D.dll. The  
| registers specific to the processor and architecture are also displayed.

| The ENVINFO section 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: 70B886AB0  
| 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  
| ICMDLINE 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
```

## Storage Management (MEMINFO)

Following on from the previous sections is the MEMINFO section, giving information from the memory manager component. See Chapter 2, “Understanding the Garbage Collector,” on page 7 for details about how the memory manager component works.

This part of the file gives various storage management values, including the free space in heap, the size of current heap, and details on other internal memory that the JVM is using. The example below shows some typical output.

```
-----
| 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
-----
```

## Locks, monitors, and deadlocks (LOCKS)

Here is an example of the LOCKS component part of a Windows 32-bit 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 (gained 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 simple deadlock test program where two threads “DeadLockThread 0” and “DeadLockThread 1” were attempting to synchronize (java keyword) on two java/lang/Integers incompatibly.

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.

```
-----
| 0SECTION      LOCKS subcomponent dump routine
| =====
```

## interpreting a Javadump

```
| 1LKPOOLINFO    Monitor pool info:  
| 2LKPPOOLTOTAL Current total number of monitors: 2  
  
| 1LKMONPOOLDUMP Monitor Pool Dump (flat & inflated object-monitors):  
| 2LKMONINUSE      sys_mon_t:0x00039B40 infl_mon_t: 0x00039B80:  
| 3LKMONOBJECT     java/lang/Integer@004B22A0/004B22AC: Flat locked by "DeadLockThread 1"  
|                           (0x41DAB100), entry count 1  
| 3LKWAITERQ       Waiting to enter:  
|                   "DeadLockThread 0" (0x41DAAD00)  
| 2LKMONINUSE      sys_mon_t:0x00039B98 infl_mon_t: 0x00039BD8:  
| 3LKMONOBJECT     java/lang/Integer@004B2290/004B229C: Flat locked by "DeadLockThread 0"  
|                           (0x41DAAD00), entry count 1  
| 3LKWAITERQ       Waiting to enter:  
|                   "DeadLockThread 1" (0x41DAB100)  
  
| 1LKREGMONDUMP   JVM System Monitor Dump (registered monitors):  
| 2LKREGMON        Thread global lock (0x00034878): <unowned>  
| 2LKREGMON        Windows native HEAP lock lock (0x000348D0): <unowned>  
| 2LKREGMON        NLS hash table lock (0x00034928): <unowned>  
| 2LKREGMON        portLibrary_j9sig_async_monitor lock (0x00034980): <unowned>  
| 2LKREGMON        Hook Interface lock (0x000349D8): <unowned>  
| < lines removed for brevity >  
  
=====  
| 1LKDEADLOCK Deadlock detected !!!  
-----  
  
| 2LKDEADLOCKTHR Thread "DeadLockThread 1" (0x41DAB100)  
|   is waiting for:  
|   sys_mon_t:0x00039B98 infl_mon_t: 0x00039BD8:  
|   java/lang/Integer@004B2290/004B229C:  
|   which is owned by:  
| 2LKDEADLOCKTHR 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:  
| 2LKDEADLOCKTHR Thread "DeadLockThread 1" (0x41DAB100)
```

## Threads and stack trace (THREADS)

For the application programmer, one of the most useful pieces of a Javadump is the THREADS section. This section shows a complete list of Java threads that are alive.

A thread is alive if it has been started but not yet stopped. 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, sys_thread_t:0x0003659C, state:R, native ID:0x00000C78) prio=5  
  at com/ibm/misc/SignalDispatcher.waitForSignal(Native Method)  
  at com/ibm/misc/SignalDispatcher.run(SignalDispatcher.java:84)
```

The properties of a thread are name, identifier, JVM data structure address, current state, native thread identifier, and priority. A large value for priority means that the thread has a high priority.

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 synchronized method of an object locked by another thread has been called
- 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.

There is a stack trace below each Java thread. A stack trace is a representation of the hierarchy of Java method calls made by the thread. The example below is taken from the same Javadump as used in the LOCKS example and two threads ("DeadLockThread 0" and "DeadLockThread 1") are both in Blocked state. The application code path that resulted in the deadlock between "DeadLockThread 0" and "DeadLockThread 1" can clearly be seen.

```
-----
THREADS subcomponent dump routine
=====

Current Thread Details
-----

All Thread Details
-----

Full thread dump J9SE VM (J2RE 5.0 IBM J9 2.3 Windows XP x86-32 build 20051012_03606_1HdSMR,
native threads):
"DestroyJavaVM helper thread" (TID:0x41508A00, sys_thread_t:0x00035EAC, state:CW,
native ID:0x00000F54) prio=5
"JIT Compilation Thread" (TID:0x41508E00, sys_thread_t:0x000360FC, state:CW, native ID:0x000000BC)
    prio=11
"Signal Dispatcher" (TID:0x41509200, sys_thread_t:0x0003659C, state:R, native ID:0x00000C78) prio=5
    at com/ibm/misc/SignalDispatcher.waitForSignal(Native Method)
    at com/ibm/misc/SignalDispatcher.run(SignalDispatcher.java:84)
"DeadLockThread 0" (TID:0x41DAAD00, sys_thread_t:0x42238A1C, state:B, native ID:0x0000073C) prio=5
    at Test$DeadlockThread0.SyncMethod(Test.java:112)
    at Test$DeadlockThread0.run(Test.java:131)
"DeadLockThread 1" (TID:0x41DAB100, sys_thread_t:0x42238C6C, state:B, native ID:0x00000738) prio=5
    at Test$DeadlockThread1.SyncMethod(Test.java:160)
    at Test$DeadlockThread1.run(Test.java:141)
```

## Classloaders and Classes (CLASSES)

See Chapter 3, "Understanding the class loader," on page 29 for information about the parent-delegation model.

The classloader (CLASSES) section includes:

- Classloader summaries. The defined class loaders and the relationship between them.
- Classloader loaded classes. The classes that are loaded by each classloader.

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.

## interpreting a Javadump

The example below 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. 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 table below details environment variables specifically concerned with Javadump production.

Table 10. Environment variables used to produce java dumps

Environment Variable	Usage Information
<b>DISABLE_JAVADUMP</b>	Setting <b>DISABLE_JAVADUMP</b> to true is the equivalent of using <b>-Xdump:java:none</b> and stops the default production of javadumps.
<b>IBM_JAVACOREDIR</b>	The default location into which the Javacore will be written. See "The location of the generated Javadump" on page 203.
<b>JAVA_DUMP_OPTS</b>	Use this environment variable to control the conditions under which Javadumps (and other dumps) are produced. See Chapter 26, "Using core (system) dumps," on page 217 for more information.

## Javadump Environment Variables

*Table 10. Environment variables used to produce java dumps (continued)*

Environment Variable	Usage Information
<b>IBM_JAVADUMP_OUTOFMEMORY</b>	By setting this environment variable to false, you disable Javadumps for an OutOfMemory condition.

## Javadump Environment Variables

---

## Chapter 25. Using Heapdump

This chapter describes:

- “Summary of Heapdump”
- “Information for users of previous releases of Heapdump”
- “Getting Heapdumps”
- “Location of the generated Heapdump” on page 214
- “Producing a Heapdump using jdmpview” on page 214
- “Available tools for processing Heapdumps” on page 215
- “Using verbose:gc to obtain heap information” on page 215
- “Environment variables and Heapdump” on page 215

---

### Summary of 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 file (using the phd 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.

---

### 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. See “Available tools for processing Heapdumps” on page 215. 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 Chapter 28, “Using the dump formatter,” on page 225.

---

### Getting Heapdumps

You can generate a Heapdump from a running JVM in these ways:

- Explicit generation
- JVM-triggered generation

Using the default dump agents provided, when the Java heap is exhausted (that is, the OutOfMemory condition is encountered and the resulting exception is not caught or handled by the application), JVM-triggered generation of a Heapdump occurs. See Chapter 23, “Using dump agents,” on page 195 for more in depth information.

Heapdumps can be generated in other situations by use of **-Xdump:heap** to generate dumps based on specific events (for example, a user-generated signal), or programmatically by use of the com.ibm.jvm.Dump.Heapdump() method from within application code. You can also use the JVMRI to request a Heapdump from a loaded agent.

## Enabling a Heapdump

To display on JVM startup the conditions (if any) that will generate a Heapdump (or jadump or systemdump), you can use **-Xdump:what**. See Chapter 23, “Using dump agents,” on page 195 for more information.

Environment variables can also affect the generation of Heapdumps (although this is a deprecated mechanism). See “Environment variables and Heapdump” on page 215 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. See “Available tools for processing Heapdumps” on page 215. However, it is sometimes useful to have an immediately readable view of the heap. You can obtain this view by using the **opts=** stanza with **-Xdump:heap** (see Chapter 23, “Using dump agents,” on page 195) or by the existence of an environment variable:

- **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**

---

## Location of the generated Heapdump

The JVM checks each of the following locations for existence and write-permission, then stores the Heapdump in the first one that is available.

- The location that is specified using the **file** suboption on the triggered **-Xdump:heap** agent.
- The location that is specified by the **IBM\_HEAPDUMPPDIR** environment variable, if set. (**\_CEE\_DMPTARG** on z/OS)
- The current working directory of the JVM processes.
- The location that is specified by the **TMPDIR** environment variable, if set.
- The **/tmp** directory. On Windows, **C:\temp**.

Enough free disk space must be available for the Heapdump file to be written correctly.

The generated Heapdump will have a name of the form:

**heapdump.%Y%m%d.%H%M%S.%pid.phd**

where:

- **%pid** is the process ID
- **.%Y%m%d.%H%M%S** is the date and time

**Notes:**

1. If “Classic” Heapdump is enabled, the name of the Heapdump will end in **txt** rather than **phd**.
2. You can override the standard names by use of the **label=** parameter. See Chapter 23, “Using dump agents,” on page 195 for more information.

---

## Producing a Heapdump using jdmpview

Before the direct production of phd files using Heapdump, you could produce them with the **hd** command from within jdmpview. For information on the use of the **hd** command, see Chapter 28, “Using the dump formatter,” on page 225.

## Available tools for processing Heapdumps

There are several tools available for Heapdump analysis, typically downloadable from the Web. Further details of the range of available tools can be found at <http://www.ibm.com/support/docview.wss?uid=swg24009436>

## Using verbose:gc to obtain heap information

Use the verbose: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, “Understanding the Garbage Collector,” 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 table below details environment variables specifically concerned with Heapdump production.

*Table 11. Environment variables used to produce heap dumps*

Environment Variable	Usage Information
<b>IBM_HEAPDUMP</b> <b>IBM_HEAP_DUMP</b>	Setting either of these to any value (such as true) enables heap dump production by means of signals.
<b>IBM_HEAPDUMPPDIR</b>	The default location into which the Heapdump will be written. See “Location of the generated Heapdump” on page 214.
<b>JAVA_DUMP_OPTS</b>	Use this environment variable to control the conditions under which Heapdumps (and other dumps) are produced. See Chapter 26, “Using core (system) dumps,” on page 217 for more information .
<b>IBM_HEAPDUMP_OUTOFMEMORY</b>	By setting this environment variable to false, you disable Heapdumps for an OutOfMemory condition.
<b>IBM_JAVA_HEAPDUMP_TEST</b>	Use this environment variable to cause the JVM to generate both matching phd and text versions of Heapdumps. Equivalent to <b>opts=PHD+CLASSIC</b> on the <b>-Xdump:heap</b> stanza.
<b>IBM_JAVA_HEAPDUMP_TEXT</b>	Use this environment variable to cause the JVM to generate a text (human readable) Heapdump. Equivalent to <b>opts=CLASSIC</b> on the <b>-Xdump:heap</b> stanza.

## **Heapdump Environment Variables**

---

## Chapter 26. Using core (system) dumps

The JVM can generate native system dumps, also known as core dumps, under configurable conditions. Core dumps are platform-specific and normally quite large. The tools used to analyze core dumps are also platform-specific. (For example, windbg on Windows and gdb on Linux.)

Chapter 28, “Using the dump formatter,” on page 225 describes a cross platform tool to analyze core dumps at the JVM level rather than the platform level.

This chapter contains the following sections:

- “Overview”
- “Defaults”
- “Environment variables and core dumps” on page 218
- “Platform-specific variations” on page 219

---

### Overview

The JVM can produce core dumps in response to specific events. Dump agents are the primary method for controlling the generation of core dump. See Chapter 23, “Using dump agents,” on page 195 for more information on dump agents.

To maintain backwards compatibility, the JVM supports the use of environment variables for core dump initiation. See “Environment variables and core dumps” on page 218 for more information.

---

### Defaults

This section describes the default agents for producing core dumps when using the JVM. Using the **-Xdump:what** option shows the following core (system) dump agent:

```
Registered dump agents
-----
dumpFn=doSystemDump
events=gpf+abort
filter=
label=C:\sdk\jre\bin\core.%Y%m%d.%H%M%S.%pid.dmp
range=1..0
priority=999
request=serial
opts=
```

This output shows that by default a core 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 UNIX)

---

## Environment variables and core dumps

The **-Xdump** option on the command line is the preferred method for generating core dumps for cases where the default settings are not enough. You can also produce core dumps using the **JAVA\_DUMP\_OPTS** environment variable. If you set the **JAVA\_DUMP\_OPTS** environment variable, default dump agents will be 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="ONcondition(dumptype,dumptype),ONcondition(dumptype,...),..."
```

where:

- *condition* can be:
  - ANYSIGNAL
  - DUMP
  - ERROR
  - INTERRUPT
  - EXCEPTION
  - OUTOFCMEMORY
- and *dumptype* can be:
  - ALL
  - NONE
  - JAVADUMP
  - SYSDUMP
  - HEAPDUMP
  - CEEDUMP (z/OS specific)

**JAVA\_DUMP\_OPTS** is parsed by taking the first (leftmost) occurrence of each condition, so duplicates are ignored. That is,

```
ONERROR(SYSDUMP),ONERROR(JAVADUMP)
```

creates system dumps for error conditions. Also, the **ONANY SIGNAL** condition is parsed before all others, so

```
ONINTERRUPT(NONE),ONANY SIGNAL(SYSDUMP)
```

has the same effect as

```
ONANY SIGNAL(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 or heap dump has been taken, before anything else.

If the **OUTOFCMEMORY** condition is used, it overrides the **IBM\_HEAPDUMP\_OUTOFCMEMORY** and **IBM\_JAVADUMP\_OUTOFCMEMORY** settings and takes the prescribed dumps whenever an out-of-memory exception is thrown (even if it is handled).

## Platform-specific variations

The variations discussed below are largely concerned with the variations involved in the use of the **JAVA\_DUMP\_OPTS** environment variable and apply to Javadumps and Heapdumps as well as system dumps.

Triggering conditions can be mapped to different signals on different platforms, and some signals are recognized on some platforms but not on others. Table 12 shows the mapping across platforms. If the JVM receives a signal that it does not recognize (that is, it is not mapped to a condition as listed in the table), the default operating system action for that signal is taken. Usually the signal is ignored.

*Table 12. Signal mappings on different platforms*

	<b>z/OS</b>	<b>Windows</b>	<b>Linux/AIX</b>
<b>EXCEPTION</b>	SIGTRAP		SIGTRAP
	SIGILL	SIGILL	SIGILL
	SIGSEGV	SIGSEGV	SISEGV
	SIGFPE	SIGFPE	SIGFPE
	SIGBUS		SIGBUS
	SIGSYS		
	SIGXCPU		SIGXCPU
	SIGXFSZ		SIGXFSZ
<b>INTERRUPT</b>	SIGINT	SIGINT	SIGINT
	SIGTERM	SIGTERM	SIGTERM
	SIGHUP		SIGHUP
<b>ERROR</b>	SIGABRT	SIGABRT	SIGABRT
<b>DUMP</b>	SIGQUIT		SIGQUIT
		SIGBREAK	

If a signal is not handled by the JVM, the operating system takes its default action for that signal. In the case of an EXCEPTION type signal, it is most likely to produce a system dump.

## **z/OS**

The full syntax for **JAVA\_DUMP\_OPTS** on z/OS can specify CEEDUMPs. See “Environment variables and core dumps” on page 218.

If **CEEDUMP** is specified, an LE CEEDUMP is produced for the relevant conditions, after any SYSDUMP 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.

Under z/OS, you can change the behavior of LE by setting the **\_CEE\_RUNOPTS** environment variable (for details refer to the *LE Programming Reference*). 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.

Dumps are produced in the following form:

## JVM dump initiation - z/OS

- **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 current directory, or as determined by the setting of `_CEE_DMPTARG` as: CEEDUMP.%Y%m%d.%H%M%S.%pid.
- **HEAPDUMP:** A reference graph of the heap is written to a file named `heapdump.%Y%m%d.T%H%M%S.phd`. See Chapter 25, “Using Heapdump,” on page 213 for more information.
- **JAVADUMP:** In the same directory as CEEDUMP, or standard JAVADUMP directory as: `javacore.%Y%m%d.%H%M%S.%pid.txt`.

## Windows

Dumps are produced in the following form:

- **SYSDUMP:** Output is written to a file named `core.%Y%m%d.%H%M%S.%pid.dmp` into the same directory that is used for JAVADUMP.
- **JAVADUMP:** Output is written to a file named `javacore.%Y%m%d.%H%M%S.%pid.txt`. See Chapter 24, “Using Javadump,” on page 203 for more information.
- **HEAPDUMP:** A heap graph is written to a file named `heapdump.%Y%m%d.%H%M%S.%pid.phd`. See Chapter 25, “Using Heapdump,” on page 213 for more information.

## AIX and Linux

Dumps are produced in the following form:

- **SYSDUMP:** Output is written to a file named `core.%Y%m%d.%H%M%S.%pid.dmp` into the same directory that is used for JAVADUMP.
- **JAVADUMP:** Output is written to a file named `javacore.%Y%m%d.%H%M%S.%pid.txt`. See Chapter 24, “Using Javadump,” on page 203 for more information.
- **HEAPDUMP:** A heap graph is written to a file named `heapdump.%Y%m%d.%H%M%S.%pid.phd`. See Chapter 25, “Using Heapdump,” on page 213 for more information.

---

## Chapter 27. Using method trace

Method trace is a powerful and free tool that allows you to trace methods in any Java code. You do not have to add any hooks or calls to existing code. Run the JVM with method trace turned on and watch the data that is returned. Using method trace provides a complete (and potentially large) diagnosis of code paths inside your application and also inside 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

Use method trace to debug and trace application code and the system classes provided with the JVM.

Method trace is part of the larger 'JVM trace' package. JVM trace is described in Chapter 33, "Tracing Java applications and the JVM," on page 283.

This chapter describes the basic use of trace. When you feel comfortable using trace, see Chapter 33, "Tracing Java applications and the JVM," on page 283 for more detailed information.

---

### Running with method trace

Control method trace by using the command-line option **-Xtrace:<option>**.

If you want method trace to be formatted, set two trace options:

- **-Xtrace:print** — set this option to 'mt' to invoke method trace.
- **-Xtrace:methods** — set this option to decide what to trace.

The first property is only a constant: **-Xtrace:print=mt**

Use the **methods** parameter to control what is traced. To trace everything, set it to **methods=\*.\***. This is not recommended because you are certain to be overwhelmed by the amount of output.

The **methods** parameter is formally defined as follows:

**-Xtrace:methods=[[!]method\_spec[,...]],print=mt**

Where **method\_spec** is formally defined as:

**{\*|[\*]classname[\*]}.{\*|[\*]methodname[\*]}[()]**

Note that

- The delimiter between parts of the package name is a forward slash, '/', even on platforms like Windows that use a backward slash as a path delimiter.
- The "!" in the **methods** parameter is a NOT operator that allows you to tell the JVM not to trace the specified method or methods. 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".

## Running with method trace

- The parentheses, (), that are in the method\_spec define whether or not to trace method parameters.
- 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`
- On UNIX platforms it may 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"`

---

## Examples of use

- **Tracing entry and exit of all methods in a given class:**  
`-Xtrace:{methods=ReaderMain.*,java/lang/String.*}`
- **Tracing entry, exit and input parameters of all methods in a class:**  
`-Xtrace:methods=ReaderMain.*()`
- **Tracing all methods in a given package:**  
`-Xtrace:methods=com/ibm/socket/*.*()`
- **Multiple method trace:**  
`-Xtrace:methods={Widget.*(),common/Gauge.*}`

This traces all method entry, exit, and parameters in the Widget class and all method entry and exit in the Gauge package.

- **Using the ! operator**  
`-Xtrace:methods={ArticleUI.*,!ArticleUI.get*}`

This traces all methods in the class ArticleUI except those beginning with "get".

---

## Where does the output appear?

In this simple case, output appears on the 'stderr'. If you want to store your output, redirect this stream to a file. You can also write method trace to a file directly, as described in "Advanced options."

---

## Advanced options

The use of method trace described above forces a formatted version of the output to be printed to the console, however, it can be rather slow. To work around this, you can make the method trace output appear in a compressed binary file and thus minimize its impact on performance. The binary data can be directed to a file using the **-Xtrace:output** option detailed in Chapter 33, "Tracing Java applications and the JVM," on page 283.

The output option will need to be combined with the required method specifications and mt component tracing options. This binary file will need to be formatted using the com.ibm.jvm.format.TraceFormat tool provided with the IBM Virtual Machine for Java. You can also route traces to your own plug-in agent and process it at will (see Chapter 33, "Tracing Java applications and the JVM," on page 283).

## Real example

```
java-Xtrace:methods={ReaderMain.*(),ConferenceUI.*()},print=mt ReaderMain
```

Results:

```
| 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.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
| 10:02:42.296 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.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). This data is fundamental because every JVM thread has its own **execenv**. Hence, 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 actually collects and emits the data.

## **Method trace - examples of use**

- The remaining fields show whether a method is being entered (>) or left (<), followed by details of the method.

---

## Chapter 28. Using the dump formatter

This chapter tells you how to use the dump formatter. It contains these topics:

- “What the dump formatter is”
- “Problems to tackle with the dump formatter” on page 226
- “Supported commands” on page 227
- “Example session” on page 232

---

### What the dump formatter is

You can run the dump formatter on one platform to work with dumps from another platform. For example, you can look at z/OS dumps on a Windows platform.

The dump formatter consists of:

#### jextract

When a dump is available, you can invoke the jextract utility. jextract produces an xml file that you can use together with the original dump to diagnose problems. jextract is in the directory  `sdk\jre\bin`. To invoke jextract, at a command prompt type:

```
jextract dumpfilename
```

By default, jextract sends its output to a file called `dumpfilename.jar` in the current directory. This file is a compressed file that contains:

- The dump
- xml produced from the dump, containing details of useful JVM internal information
- Other files that can help in diagnosing the dump (such as trace entry definition files)

Normally, you would send the jar file to IBM for problem diagnosis.

Chapter 12, “Submitting data with a problem report,” on page 89 tells you how to do that.

Unzip the zip file into a unique folder using `unzip -d dir`. Using a unique folder ensures that you do not overwrite any existing files on your system. You are also advised to run jdmpview from that new folder.

The contents of the jar file produced and the contents of the xml are subject to change, so you are advised not to design tools based on the contents of these. Preferably, you run jextract on the same system as the one on which the dump was produced. However, you can run jextract on a system that has the same version of the JRE as the system on which the dump was produced.

You can use the `-nozip` option to bypass packaging of the data into a compressed file and a `-f filename` option to direct the output to a location other than the default.

## What the dump formatter is

### jdumpview

**jdumpview** is a launcher for the main method of the java class J9JVMConsole that is contained in the sdk/jre/lib/ext/jdumpview.jar. To invoke jdumpview, from a command prompt type:

```
jdumpview [[-d]dumpfilename] [-wworkdir] [-ooutput]
```

where

- *dumpfilename* is a dumpfile
- *workdir* is a writable directory
- *output* is an output file (typical format *file:x:\myfile*)

Typical usage is `jdumpview my.dmp`. The J9JVMConsole class opens and verifies the `my.dmp` file (which it recognizes as a dump file) and the associated xml file (`my.dmp.xml`).

After `jdumpview` processes the arguments with which it was launched, it displays the message Ready.... When you see this message, you can start invoking commands on `jdumpview`. You can run an unlimited number of `jdumpview` sessions at the same time.

You can significantly improve the performance of `jdumpview` 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 have to invoke `jdumpview` using the `-Xmx` option to increase the maximum heap available to `jdumpview`:

```
jdumpview -J-Xmx<n>
```

To pass command-line arguments to the JVM, you must prefix them with `-J`. For more information on using `-Xmx`, see Appendix D, “Command-line options,” on page 361.

---

## Problems to tackle with the dump formatter

Dumps including a JVM can arise:

- With the JVM in control (that is, when you specify the `-Xdump` option on the command line)
- From handled events (such as an `OutOfMemory` exception)
- When the JVM is not in control (such as user-initiated dumps)

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 could have been taken while the VM was in an inconsistent state; the following option ensures that the JVM (and the Java heap) is in a safe state before taking any system dump:

```
-Xdump:system:defaults:request=exclusive+prepwalk
```

Setting this option adds a non-trivial overhead to taking a system dump; this overhead could cause problems in rare situations, so it is not enabled by default.

`jdumpview` is most useful in diagnosing customer-type problems and problems with the J2SE class libraries. A typical scenario is `OutOfMemory` errors in customer applications.

For problems involving gprs, ABENDS, SIGSEVs, and similar problems, you obtain more information by using the system debugger (windbg, gdb) with the dump file. However, `jdumpview` can still provide useful information when used alone.

## Supported commands

This section describes the commands available in jdumpview. 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 jdumpview session.

- **cmds**

*Availability:* `always`

Displays the available commands at any point during the jdumpview 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 not available 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, it displays specific help on a command. For example, `help dis os` produces help information regarding the `help 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 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`.

- **display proc**

*Short form:* `dis proc`

*Availability:* `after set dump has run`

Displays information about the process or processes found in the dump. It shows the command line that launched the process, thread information, environment variables, and loaded modules.

- **display sym**

*Short form:* `dis sym`

*Availability:* `after set dump has run`

As part of its processing, jdumpview creates symbols allowing 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**

## Dump formatter - supported commands

*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. The variations of set are covered below.

set without any parameters shows what 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 executed 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. If you want to analyze another dump, you must start a new jdmpview session.

- **set metadata**

*Short form: s meta*

*Availability: after set dump has run*

Starts the reading of the xml file produced by jextract, causes the xml file to be parsed, and gives assorted details about the underlying 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 of it will 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. Usually, 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 launching 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).

- **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).

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.

- **display sys**

*Short form: d sys*

*Availability: after set metadata has run*

Gives information about the dump and the JVM.

## Commands for analysing the memory

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. These commands do not function until after the set dump command has been successfully issued.

- **display mmap**

*Short form: dis mmap*

*Availability: after set dump has run*

When a dump is opened (set dump), jdumpview establishes a mapping of virtual memory ranges held in the dump to their location in the dump file. In doing this, it creates an internal map which is then used by the rest of the commands to access memory within the dump. Dis mmap allows this mapping to be displayed and allows the user to see 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) and dis mmap, besides showing the ranges and their offsets, also shows the asid to which the memory range belongs. Be aware that areas of memory that appear contiguous (or even overlap) according to the memory map are almost certainly not contiguous and will have different asids.

- **display mem**

*Short form: dis mem*

*Availability: after set dump has run*

Displays memory within the dump. The syntax is:

dis mem <address>[,<numbytes>]

where <address> is the hex address to display (it can be preceded by 0x) and <numbytes> (defaults to 256) is the number of bytes to display.

## Dump formatter - supported commands

- **find**

*Short form: fn*

*Availability: after set dump has run*

Looks for strings and hex values within the dump memory. The syntax is

`Find pattern[,<start>][,<end>][,<boundary>][,<count>][,<limit>]`

The `<start>` parameter controls where to start the search, `<end>` where to end it, `<boundary>` what byte boundary should be used, `<count>` how many bytes of memory should be displayed when a hit is encountered, and `<limit>` the limit of occurrences to display.

## Commands for 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*

Without a `<classname>` specified, produces a list of all the known classes together with their instance size and (if `dis os` has run) a count of the instances associated with that class. 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 displayed. For classes such as `[char`, where the `'` indicates that this is 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 on the composition of the specified class. It displays the methods, fields, and statics associated with the class with other information. DO NOT confuse it 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.

## Commands for 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).

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. The information displayed about an object is produced along with the stored details for its class.

## Commands for working with 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, it 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 machine and the host to succeed.

### Commands for 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 jdmpview with the name of a dump }

jdmpview 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 d12.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) << jdmpview
                                                               is ready to accept user input

{ the output produced by h (or help) is illustrated below - "help
```

## Dump formatter - example session

<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
```

## Dump formatter - example session

```
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 jdumpview session)
HELP * (Provides generic and specific help)
```

## Dump formatter - example session

```
==== 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: 0x41fcbd00 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
```

## Dump formatter - example session

```
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: 0xb64 (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 | p1.[4...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 ... }
```

## Dump formatter - example session

```
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;
```

## Dump formatter - example session

```
                                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=0x41e0fcc
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=0x41e0fddc

Ready .....
```

## Dump formatter - example session

```
{ 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(J)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	.....
00083ccb: 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]
```

## Dump formatter - example session

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 endianess 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,ffffffffffff,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:

## Dump formatter - example session

Findp pointer[,start][,end][,boundary][,count][,limit]

Examples are:

Findp 0x10000 -

this would search all available memory for the pointer 0x10000

Note: jdumpview adjusts the input command to match the endianess 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 endianess 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 ....

## Dump formatter - example session

---

## Chapter 29. JIT problem determination

The Just-In-Time compiler (JIT) is tightly bound to the JVM, but is not part of it. The JIT converts Java bytecodes, which are interpreted by the JVM at run time and execute slowly, into native code, which is understood by the processor and executes quickly.

Occasionally, valid bytecodes might compile into invalid native code, causing the Java program to fail. By determining whether the JIT is faulty and, if so, *where* it is faulty, you can provide valuable help to the Java service team.

This chapter describes how you can determine with reasonable certainty whether your problem is JIT-related. This chapter also suggests some possible workarounds and debugging techniques for solving JIT-related problems:

- “Disabling the JIT”
- “Selectively disabling the JIT”
- “Locating the failing method” on page 244
- “Identifying JIT compilation failures” on page 245
- “Performance of short-running applications” on page 246

---

### Disabling the JIT

The JIT is enabled by default, but 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 JIT threshold, the method is compiled and executed natively.

The call count mechanism spreads compilation of methods throughout the life of an application, giving higher priority to methods that are used most frequently. Some infrequently used methods might never be compiled at all. As a result, when a Java program fails, the problem might be in the JIT, or 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 disabled): specify the `-Xint` option, and remove the `-Xjit` option (and accompanying JIT parameters, if any) when you run the JVM. If the failure still occurs, the problem is most likely in the JVM rather than the JIT. (Do not use the `-Xint` and the `-Xjit` options together.)

Running the Java program with the JIT disabled leads to one of the following:

- The failure remains. The problem is, therefore, not in the JIT. Do not read further in this chapter. In some cases, the program might start failing in a different manner; nevertheless, the problem is not related to the JIT.
- The failure disappears. The problem is most likely, although not definitely, in the JIT.

---

### Selectively disabling the JIT

If the failure of your Java program appears to come from a problem within the JIT, you can try to narrow down the problem further.

## JIT problem determination

The JIT optimizes methods at various optimization levels; that is, 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 parameters, you can control the optimization level at which methods are optimized, and determine whether the optimizer is at fault and, if it is, which optimization is problematic.

JIT parameters are specified 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, while enabling verbose output from the JIT, and making it generate native code without performing any of the optimizations listed in “How the JIT optimizes code” on page 36.

The JIT parameters give you a powerful tool that enables you to determine the location of a JIT problem; whether it is in the JIT itself or in some Java code that causes the JIT to fail. In addition, when you have identified a problem area, you are automatically given a workaround so that you can continue to develop or deploy code while losing only a fraction of JVM performance.

The first JIT parameter to try is **count=0**, which sets the JIT threshold to zero and effectively causes the Java program to be run in purely compiled mode.

If the failure still occurs, add **disableInlining** to the command line. With this parameter set, the JIT is prohibited from generating larger and more complex code in an attempt to perform aggressive optimizations.

If the failure persists, try decreasing JIT optimization levels. The various optimization levels are:

1. scorching
2. veryHot
3. hot
4. warm
5. cold
6. noOpt

Run the Java program with:

```
-Xjit:count=0,disableInlining,optLevel=scorching
```

Try each of the optimization levels in turn, and record your observations. If one of these settings causes your failure to disappear, you have a quick workaround that you can use while the Java service team analyzes and fixes the JIT problem. If you can remove **disableInlining** from the JIT parameter list (that is, if removing it does not cause the failure to reappear), do so to improve performance.

---

## Locating the failing method

When you have arrived at the lowest optimization level at which the JIT must compile methods to trigger the failure, you can try to find out which part of the Java program, when compiled, causes the failure. You can then instruct the JIT to limit the workaround to a specific method, class, or package, allowing the JIT to compile the rest of the program as it normally would. If the failure occurs with

**optLevel=noOpt**, you can also instruct the JIT to not compile the method or methods that are causing the failure (thus avoiding it).

To locate the method that is causing the failure, follow these steps:

1. Run the Java program with the JIT parameters **verbose** and **vlog=filename**. With these parameters, the JIT reports its progress, as it compiles methods, in a verbose log file, 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 JIT in the steps below, so you can edit them out of 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 respectively the first and the last methods in the limit file that should be compiled. This parameter causes the JIT to compile 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. Repeat this step with a smaller range if the program still fails.

The recommended number of lines to select from the limit file in each repetition is half of the previous selection, so that this step is essentially a binary search for the failing method.

3. 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. Select methods not selected in the previous iteration and repeat the previous step to see if the program fails again.
4. Repeat the last two steps, as many times as necessary, to find the minimum number of methods that must be compiled to trigger the failure. Often, you can reduce the file to a single line.

When you have obtained a workaround and located the failing method, you can limit the workaround to the failing method. For example, if the method `java/lang/Math.max(II)I` causes the program to fail when compiled with **optLevel=hot**, you can run the program with:

```
-Xjit:{java/lang/Math.max(II)I}{optLevel=warm,count=0}
```

which tells the JIT to compile only the troublesome method at an optimization level of "warm", but compile all other methods normally.

If a method fails when it is 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

If the JVM crashes, and you can see that the failure has occurred in the JIT library (`j9jit23.dll` on Windows, or `libj9jit23.so` on other platforms), the JIT might have failed during an attempt to compile a method.

To see if the JIT is crashing in the middle of a compilation, use the **verbose** option with the following additional settings:

```
-Xjit:verbose={compileStart|compileEnd}
```

## JIT problem determination

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 244). If excluding the method prevents the crash, you have an excellent workaround that you can use while the service team correct your problem.

---

## Performance of short-running applications

The IBM JIT is tuned for long-running applications typically used on a server. So, if the performance of short-running applications is worse than expected, try the **-Xquickstart** command-line parameter (refer to the **-Xquickstart** option in “Nonstandard command-line options” on page 364), especially for those applications in which execution time is not concentrated into a small number of methods.

Also try adjusting the JIT threshold (using trial and error) for short-running applications to improve performance. Refer to “Selectively disabling the JIT” on page 243.

---

## Chapter 30. Garbage Collector diagnostics

This chapter describes how to diagnose the garbage collection operation. The topics that are discussed in this chapter are:

- “How do the garbage collectors work?”
- “Common causes of perceived leaks”
- “Basic diagnostics (-verbose:gc)” on page 248
- “Advanced diagnostics” on page 257
- “-Xtgc tracing” on page 260
- “Native memory use by the JVM” on page 263

---

### How do the garbage collectors work?

Read Chapter 2, “Understanding the Garbage Collector,” on page 7 to get a full understanding of the Garbage Collector. A short introduction to the Garbage Collector is given here.

The IBM Virtual Machine for Java 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. Garbage collection identifies and frees previously allocated storage that is no longer in use. When this process is complete, the memory manager retries the allocation that it could not previously satisfy.

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 22.

---

### 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. So, 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 effectively attach your object to a static reference that is in the listener. Your object cannot be collected while the listener is active. You must explicitly uninstall a listener when you have finished using the object to which you attached it.

### 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 data

This 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 an application across the JNI interface have a reference to them that is held in the JNI code of the JVM. Without this reference, the Garbage Collector cannot trace live native objects. Such references must be explicitly cleared by the native code application before they can be collected. See the JNI documentation on the Sun website ([java.sun.com](http://java.sun.com)) for more information.

### Premature expectation

You instantiate a class, finish with it, tidy up all listeners, and so on. You have a finalizer in the class, and you use that finalizer to report that the finalizer has been called. On all the later garbage collection cycles, your finalizer is not called. It seems that your unused object is not being collected and that a memory leak has occurred, but this is not so.

The IBM Garbage Collector does *not* collect garbage unless it needs to. It does not necessarily collect all garbage when it does run. It might not collect garbage if you manually invoke it (by using `System.gc()`). This is because running the Garbage Collector is an intensive operation, and it is designed to run as infrequently as possible for as short a time as possible.

### Objects with finalizers

Objects that have finalizers cannot be collected until the finalizer has run. Finalizers run on a separate thread, and thus their execution might be delayed, or not occur at all. This is allowed. See "How to coexist with the Garbage Collector" on page 22 for more details.

---

## Basic diagnostics (-verbose:gc)

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 invoking 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, "Understanding the Garbage Collector," on page 7 if necessary.

## Garbage collection triggered by System.gc()

Java programs can trigger garbage collections to occur manually by invoking the method `System.gc()`. -verbose:gc output produced by `System.gc()` calls is similar to:

```
<sys id="1" timestamp="Fri Jul 15 12:56:26 2005" intervalms="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" intervalms="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>
```

**<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 **intervalms** 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.

### <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 appear, 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. Normally, 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.

### <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).

### <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 251.

### <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.

### Nursery allocation failures

```
<af type="nursery" id="28" timestamp="Fri Jul 15 13:11:45 2005" intervalms="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" intervalms="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 tiltratio="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 nursery. The **id** attribute shows the index of that type of allocation failure that has occurred. **timestamp** shows a local timestamp at the time of the allocation failure, and **intervalms** shows the number of milliseconds elapsed since the previous allocation failure of that type.

#### <minimum>

Shows the number of bytes requested by the allocate that triggered the failure. Note that, following the garbage collection, freespace might drop by more than this amount, because of a possible freelist discard or TLH refresh.

#### <nursery>

Shows the status of the nursery at the time of the failure, including the percentage that was free.

**<gc>** Indicates that, as a result of the allocation failure, a garbage collection was triggered. In this case, a scavenger collection occurred. The contents of this tag are explained in detail in "Scavenger collections" on page 252.

### <nursery> and <tenured>

Show the status of the different heap areas following the handling of the allocation failure, including the percentage of each area free.

### <time>

Shows the total time taken to handle the allocation failure.

## Tenured allocation failures

Here is an example of an allocation occurring in the tenured area:

```

<af type="tenured" id="2" timestamp="Fri Jul 15 13:17:11 2005" intervalms="450.0
57">
  <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" intervalms="217.002">
    <expansion type="tenured" amount="1048576" newsizes="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>

```

All the elements of this output have the same meanings as those for an allocation failure occurring in the nursery. The only difference is the addition of an expansion tag.

## Global collections

An example of the output produced when a global collection is triggered is:

```

<gc type="global" id="5" totalid="5" intervalms="18.880">
  <compaction movecount="9282" movebytes="508064" reason="forced compaction" />
  <expansion type="tenured" amount="1048576" newsizes="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 this was a global collection (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 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.

## Garbage Collector - basic diagnostics (-verbose:gc)

### <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 appears 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 that this is not equal to the number of finalizers that were run during the collection, because finalizers are scheduled by the VM.

### <timems>

Provides information detailing, respectively, times taken for each of the mark, sweep, and compact phases, as well as the total time taken. When compaction was not triggered, the number returned is zero.

### <tenured>

Indicates the status of the tenured area following the collection. If running in generational mode, there will also be a <nursery> line output, showing the status of the active nursery area too.

## Scavenger collections

An example of the output produced when a scavenger collection is triggered is:

```
<gc type="scavenger" id="11" totalid="11" intervalms="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, and **type="scavenger"** indicates that this is a scavenger collection. The **id** attribute shows the number of this type of collection that have taken place and 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 tenure some objects when it tried to during the collection. The number affected and the total bytes represented

by these objects is shown. Additionally or alternatively, <failed type="flipped"> could have been displayed, which would indicate that the scavenger failed to flip certain objects into the survivor space.

### <flipped>

Shows the number of objects that were flipped into the survivor space during the scavenge, together with the total number of bytes flipped.

### <scavenger tiltratio="x" />

Shows the percentage that new-space is tilted by, following the post-scavenge retilt. The scavenger can redistribute memory between the allocate and survivor areas to maximize the time between scavenges and the number of objects that "die young".

### <tenured>

Shows the number of objects that were moved into the old area during the scavenge, together with the total number of bytes tenured.

### <nursery>

Shows the amount of free and total space in the nursery area following the scavenge, along with the current number of flips an object must have survived in order to be tenured.

### <time>

Shows the total time taken to perform the scavenge, in milliseconds.

A number of additional lines can be output during a scavenge. It is possible for a scavenge to fail (for example, if the nursery was excessively tilted with a full old area, and certain objects could not be copied or tenured). In this case, an additional <warning details="aborted collection" /> line is displayed.

During a scavenge, if it is not possible to tenure an object, an expansion of the tenured area might be triggered. This will be shown as a separate line of **-verbose:gc**.

During a scavenge, if remembered set overflow or scan cache overflow occurred, these will also be shown as separate lines of **-verbose:gc**.

It is also possible for the entirety of new space to be resized following a scavenge. Again, this is shown as a separate line of **-verbose:gc**.

## Concurrent garbage collection

When running with concurrent garbage collection, several additional **-verbose:gc** outputs are displayed.

### Concurrent sweep completed

```
<con event="completed sweep" timestamp="Fri Jul 15 13:52:08 2005">
  <stats bytes="0" time="0.004" />
</con>
```

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.

### Concurrent kickoff

When the concurrent mark process is triggered, the following output is produced:

## Garbage Collector - basic diagnostics (-verbose:gc)

```
<con event="kickoff" timestamp="Fri Nov 25 10:18:52 2005">
  stats tenurerefabytes="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 within 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 nursery as concurrent mark was triggered.

### Allocation failures during concurrent mark

When an allocation failure occurs during concurrent mark, either the tracing performed so far will be discarded, or it will be used during the subsequent collection. These two possibilities correspond to the "aborted" and "halted" concurrent mark events.

**Concurrent aborted:** Here is a sample of the output produced when concurrent mark is aborted:

```
<af type="tenured" id="4" timestamp="Fri Jul 15 14:08:28 2005" intervalms="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" intervalms="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:** Here is a sample output produced when concurrent mark is halted:

```
<af type="tenured" id="5" timestamp="Fri Jul 15 14:08:28 2005" intervalms="249.9
55">
  <minimum requested_bytes="32" />
  <time exclusiveaccessms="0.022" />
```

## Garbage Collector - basic diagnostics (-verbose:gc)

```
<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 actually 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 is shown:

```
<con event="collection" id="15" timestamp="Fri 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="Fri Jul 15 15:13:18 2005">
  <stats sweepbytes="0" sweepetime="0.009" connectbytes="5826560"
    connecttime="0.122" />
</con>
```

## Garbage Collector - basic diagnostics (-verbose:gc)

```
<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, a local timestamp is outputted, 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 that which actually 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

```
<sys id="6" timestamp="Fri 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="Fri Jul 15 15:57:49 2005">
    <stats sweepbytes="0" sweepetime="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>
</sys>
```

```

</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 kicked off. In this case, enough tracing had been performed for the work to be reused, so concurrent mark is halted rather than aborted. The results for final card-cleaning are also shown.

## Advanced diagnostics

The `-verbose:gc` option is the main diagnostic 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. These options are:

- `-Xdisableexplicitgc`
- `-Xgcthreads`
- `-Xconcurrentbackground`
- `-Xconcurrentlevel`
- `-Xgcworkpackets`
- `-Xclassgc`
- `-Xalwaysclassgc`
- `-Xnoclassgc`
- `-Xcompactgc`
- `-Xnocompactgc`
- `-Xcompactexplicitgc`
- `-Xnocompactexplicitgc`
- `-Xpartialcompactgc`
- `-Xnopartialcompactgc`
- `-Xenablestringconstantgc`
- `-Xdisablestringconstantgc`
- `-Xlp`
- `-Xnoloa`
- `-Xloainitial`
- `-Xloamaximum`
- `-Xcometer:<soa | loa | dynamic>`
- `-Xenableexcessivegc`
- `-Xdisableexcessivegc`

## Garbage Collector - advanced diagnostics

### -Xsoftrefthreshold

#### **-Xdisableexplicitgc**

This option converts Java application calls to `java.lang.System.gc()` into no-ops.

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, it is not always possible to remove the calls at source.

The `-Xdisableexplicitgc` parameter allows the JVM to ignore these garbage collection suggestions. Typically, system administrators would use this parameter in applications that show some benefit from its use. `-Xdisableexplicitgc` is a nondefault setting.

`-Xdisableexplicitgc` should be used only when testing had shown it to be beneficial; for example, from performance testing in conjunction with `-verbose:gc` output.

#### **-Xgcthreads**

This option sets the number of helper threads that the Garbage Collector uses for parallel operations. By default, the number is set to (number of CPUs – 1). On single CPU boxes, no helper threads run. To set it to a different number (for example 4), use `-Xgcthreads4`. The disabling of helper threads disables parallel operations, at the cost of performance, and might expose problems in this area. No advantage is gained if you increase the number of threads above the default setting; you are recommended not to do so.

#### **-Xconcurrentbackground<number>**

This option specifies the number of low priority background threads attached to assist the mutator threads in concurrent mark. The default is 1.

#### **-Xconcurrentlevel<number>**

This option 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.

#### **-Xgcworkpackets<number>**

This option 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.

#### **-Xclasssgc**

This option enables the collection of class objects only on class loader changes.

#### **-Xalwaysclasssgc**

This option enables collection of class objects at every garbage collection.

#### **-Xnoclasssgc**

This option disables collection of class objects. This switches off garbage collection of storage associated with Java classes that are no longer being used by the JVM.

**-Xcompactgc**

This option enables compaction at every garbage collection.

**-Xnocompactgc**

This option disables heap compaction.

**-Xcompactexplicitgc**

This option runs full compaction each time `System.gc()` is called.

**-Xnocompactexplicitgc**

This option means a compaction is never run when `System.gc()` is called.

**-Xpartialcompactgc**

This option enables incremental compaction.

**-Xnopartialcompactgc**

This option disables incremental compaction, and forces full compactions.

**-Xenablestringconstantgc**

This option enables strings from the string intern table to be collected.

**-Xdisablestringconstantgc**

This option prevents strings in the string intern table from being collected.

**-Xlp**

This option enables large page support. This option is available only on AIX (both PPC32 and PPC64), Windows Server 2003 (x86, AMD64, EM64T), and Linux (x86, PPC32, PPC64, AMD64, EM64T).

**-Xnoloa**

This option prevents allocation of a large object area; all objects will be allocated in the SOA.

**-Xloainitial<number>**

<number> is a floating point number between 0 and 0.95, which specifies the initial percentage of the current tenure space allocated to the large object area (LOA). The default is 0.05 or 5%.

**-Xloamaximum<number>**

<number> is a floating point number between 0 and 0.95, which specifies the maximum percentage of the current tenure space allocated to the large object area(LOA). The default is 0.5 or 50%.

**-Xconmeter:<soa | loa | dynamic>**

This option determines which area's usage, LOA or SOA, is metered and hence which allocations are taxed during concurrent mark. Using **-Xconmeter:soa** 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

## Garbage Collector - advanced diagnostics

(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.

### **-Xenableexcessivegc**

If excessive time is spent in the GC, this 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.

### **-Xdisableexcessivegc**

This option disables the throwing of an OutOfMemory exception if excessive time is spent in the GC.

### **-Xsoftrefthreshold<number>**

This option sets the number of GCs after which a soft reference will be cleared if its referent has not been marked. The default is 32, meaning that the soft reference will be cleared on the 33rd consecutive GC in which the referent has not been marked.

---

## **-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 piped 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 vmThread 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, 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". The compaction reasons are explained in detail in "Compaction phase" on page 8. 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=x000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA25D0 freelen=x00000010>
<GC(4) 13FA25E0 freelen=x00000048 -- x0000001C java/lang/String>
<GC(4) 13FA2890 freelen=x000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA2B8C freelen=x00000010>
<GC(4) 13FA2B9C freelen=x00000068 -- x0000001C java/lang/String>
<GC(4) 13FA2E8C freelen=x000002C4 -- x00000038 spec/jbb/Stock>
<GC(4) 13FA3188 freelen=x00000010>
```

A line of output is printed for every free chunk in the system, including dark matter (free chunks that are not on the free list for some reason, usually because they are too small). Each line contains the base address and the size in bytes of the chunk. If the chunk is followed in the heap by an object, the size and class name of the object is also printed.

## -Xtgc:freelist

Before a garbage collection, this trace prints information about the free list and allocation statistics since the last GC. It prints the number of items on the free list, including "deferred" entries (with the scavenger, the unused semispace is a

## Garbage Collector - -Xtgc tracing

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 GC.

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) and the work done by each. For mark, the time spent busy, stalled, and in tail is shown (in milliseconds) together with the number of work packets each thread acquired and released during marking. For sweep, the time spent busy and idle is shown (in milliseconds) together with the number of sweep chunks processed by each thread and in total (127 above). The total merge time is also shown (0ms above).

## -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 that for finalizer objects this does not mean the finalizer has been run, merely that it has been queued to 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 below:

```
{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 0
{SCAV: java/io/FileOutputStream 0 2 0 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 0
{SCAV: java/io/FileInputStream 0 1 0 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 0
{SCAV: java/lang/ref/SoftReference[] 0 2 0 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 0
```

```
{SCAV: java/io/BufferedWriter 0 4 0 0 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 0 0  
{SCAV: java/io/PrintStream 0 2 0 0 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 0 0  
{SCAV: java/lang/Thread[] 0 2 0 0 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 0 0  
{SCAV: sun/io/ByteToCharCp1252 0 1 0 0 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 0 0
```

**-Xtgc:terse**

This trace dumps the contents of the entire heap before and after a garbage collection. As such, 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 below:

```
*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[]
```

## Native memory use by the JVM

The JVM does use native memory, but, for efficiency, does not use standard stack frames. The JIT (see Chapter 5, “Understanding the JIT,” on page 35), the Interpreter (see Chapter 29, “JIT problem determination,” on page 243), and the JVM all have their own styles of stack frames. The only tool that can walk the stack is the dump formatter (see Chapter 28, “Using the dump formatter,” on page 225). The only other users of native memory are native code and some types of large native objects.

## Native code

The term “native code ” refers to native code (usually C or C++) that is compiled into a library and accessed through the JNI. Alternatively, native code can load an encapsulated JVM. Either way, the native code uses standard OS stack frames,

## **Garbage Collector - heap and native memory use by the JVM**

unless it manages the stack itself. The JVM keeps track of the portion of the stack that it uses, because it needs this information to find a set of root objects for garbage collection.

The JVM has no knowledge of and cannot control the native stack in this scenario. Growth of the native stack is not normally due to JVM code.

## **Large native objects**

On some platforms, the JVM can recognize large native objects (such as bitmaps) and keep them in native memory. A small object is placed onto the heap, which acts as an anchor for the native data (wherever it is). Clearly, such native memory tends to consist of large chunks that can grow quickly unless the owning application strictly controls the anchoring objects.

---

## Chapter 31. Class-loader diagnostics

This chapter describes 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 266

---

### Class-loader command-line options

These extended command-line options are available:

**-verbose:dynload**

This option 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
- 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 verifier. Note that this is the default in all J2SE JVMs; used on its own, this 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

---

### Class-loader runtime diagnostics

An extremely useful command-line definition is available that lets you trace the way the class loaders find and load a given class. The command-line definition is **-Dibm.cl.verbose=<name>**

For example:

```
C:\j9test>java -Dibm.cl.verbose=HelloWorld HelloWorld
```

might produce output that is similar to this:

```
ExtClassLoader attempting to find HelloWorld
ExtClassLoader using classpath C:\j9test\testjdk\ sdk\jre\lib\ext\gskikm.jar;C:\j
9test\testjdk\ sdk\jre\lib\ext\ibmjcefips.jar;C:\j9test\testjdk\ sdk\jre\lib\ext\i
bmjceprovider.jar;C:\j9test\testjdk\ sdk\jre\lib\ext\ibmjsseprovider2.jar;C:\j9te
st\testjdk\ sdk\jre\lib\ext\ibmpkcs11.jar;C:\j9test\testjdk\ sdk\jre\lib\ext\ibmpk
```

## class-loader diagnostics

```
cs11impl.jar;C:\j9test\testjdk\sdk\jre\lib\ext\indicim.jar;C:\j9test\testjdk\sdk
\jre\lib\ext\jaccess.jar;C:\j9test\testjdk\sdk\jre\lib\ext\jdmpview.jar;C:\j9tes
t\testjdk\sdk\jre\lib\ext\ldapsec.jar;C:\j9test\testjdk\sdk\jre\lib\ext\oldcertp
ath.jar
ExtClassLoader could not find HelloWorld.class in C:\j9test\testjdk\sdk\jre\lib\
ext\gskikm.jar
ExtClassLoader could not find HelloWorld.class in C:\j9test\testjdk\sdk\jre\lib\
ext\ibmjcefips.jar
ExtClassLoader could not find HelloWorld.class in C:\j9test\testjdk\sdk\jre\lib\
ext\ibmjcepovider.jar
ExtClassLoader could not find HelloWorld.class in C:\j9test\testjdk\sdk\jre\lib\
ext\ibmjssesprovider2.jar
ExtClassLoader could not find HelloWorld.class in C:\j9test\testjdk\sdk\jre\lib\
ext\ibmpkcs11.jar
ExtClassLoader could not find HelloWorld.class in C:\j9test\testjdk\sdk\jre\lib\
ext\ibmpkcs11impl.jar
ExtClassLoader could not find HelloWorld.class in C:\j9test\testjdk\sdk\jre\lib\
ext\indicim.jar
ExtClassLoader could not find HelloWorld.class in C:\j9test\testjdk\sdk\jre\lib\
ext\jaccess.jar
ExtClassLoader could not find HelloWorld.class in C:\j9test\testjdk\sdk\jre\lib\
ext\jdmpview.jar
ExtClassLoader could not find HelloWorld.class in C:\j9test\testjdk\sdk\jre\lib\
ext\ldapsec.jar
ExtClassLoader could not find HelloWorld.class in C:\j9test\testjdk\sdk\jre\lib\
ext\oldcertpath.jar
ExtClassLoader could not find HelloWorld

AppClassLoader attempting to find HelloWorld
AppClassLoader using classpath C:\test\ras
AppClassLoader found HelloWorld.class in C:\test\ras
AppClassLoader found HelloWorld
```

The sequence of the loaders output is due to the "delegate first" convention of the class loaders. In this convention, each loader checks its cache, then delegates to its parent loader. Then, if the parent returns null, the loader checks the file system or equivalent. This is the part of the process that is reported in the example above. In the command-line definition, the classname can be given as any Java regular expression. "Dic\*" will produce output on all classes whose names begin with the letters "Dic", and so on.

---

## Loading from native code

When a native library is being loaded, how the class that makes the native call is loaded determines where the loader looks to load the libraries.

- 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.

## Chapter 32. Shared classes diagnostics

This chapter describes how to diagnose problems that might occur in shared classes mode. The topics that are discussed in this chapter are:

- “Understanding shared classes diagnostics output”
- “Deploying Shared Classes” on page 270
- “Dealing with runtime bytecode modification” on page 273
- “Understanding dynamic updates” on page 275
- “Using the Java Helper API” on page 277
- “Debugging problems with shared classes” on page 279
- “Class sharing with OSGi ClassLoading framework” on page 282

### 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 option gives the most concise and simple diagnostic output on cache usage. 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 shutdown shows how many bytes were read or stored in the cache.

#### VerboseIO output

The verboseIO output is far more detailed and is used at runtime to show classes being stored and found in the cache. You enable verboseIO output by using the **verboseIO** suboption of **-Xshareclasses**. VerboseIO output provides information about the I/O activity occurring with the cache, with basic information on find and store calls. With a cold cache, you see trace like this:

```
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 and the bootstrap loader is always 0. In the trace above, you see ClassLoader 17 obeying the classloader hierarchy of asking its parents for the class. So each of its parents consequently asks the shared cache for the class; because it does not yet exist in the cache, all the find calls fail and ClassLoader 17 stores it.

## Understanding shared classes diagnostics output

After the class is stored, you see the following output:

```
| 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, with its parents asking the cache for the class first. It succeeds for the correct ClassLoader. Note that with alternative classloading frameworks, such as OSGi, the parent delegation rules are different, so you will not necessarily see this type of output.

## VerboseHelper output

You can also obtain diagnostics from the Java SharedClassHelper API using the **verboseHelper** option; see “Using the Java Helper API” on page 277. 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>

## printStats utility

The printStats utility is a suboption of **-Xshareclasses**, optionally taking a cache name using **name=<name>**. It gives summary information on the cache specified. Because it is a cache utility, the VM displays the information on the cache specified and then exits.

Here is a detailed description of what the output means:

```
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

Show the total size of the shared memory area in bytes and the free bytes remaining respectively.

### 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

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 (see “Using the Java Helper API” on page 277) used to store the classes

### Classpaths, URLs, and Tokens

Classes stored from a SharedClassURLClasspathHelper are stored with a Classpath; those stored using a SharedClassURLHelper are stored with a URL; and those stored using a SharedClassTokenHelper are stored with a Token. Most ClassLoaders (including the bootstrap and application classloaders) use a SharedClassURLClasspathHelper, so 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 (for example, a Jar is rebuilt), a new Classpath is added to the cache. Additionally, if “partitions” or “modification contexts” are used, these are associated with the Classpath, URL, and Token, and one 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 275.

### % 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 walks the entire cache from start to finish and lists its contents. It aims to give as much diagnostic information as possible and, 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 VM displays the information on the cache specified and then exits.

Each JVM that connects to the cache receives a unique ID and each entry in the output is preceded by a number indicating the JVM that wrote the data.

### Classpaths

```
1: 0x2234FA6C CLASSPATH
    C:\myJVM\sdk\jre\lib\vm.jar
    C:\myJVM\sdk\jre\lib\core.jar
    C:\myJVM\sdk\jre\lib\charsets.jar
    C:\myJVM\sdk\jre\lib\graphics.jar
    C:\myJVM\sdk\jre\lib\security.jar
    C:\myJVM\sdk\jre\lib\ibmpkcs.jar
    C:\myJVM\sdk\jre\lib\ibmorb.jar
    C:\myJVM\sdk\jre\lib\ibmcfw.jar
    C:\myJVM\sdk\jre\lib\ibmorbapi.jar
    C:\myJVM\sdk\jre\lib\ibmjcefw.jar
```

## Understanding shared classes diagnostics output

```
C:\myJVM\ sdk\jre\lib\ibmjgssprovider.jar  
C:\myJVM\ sdk\jre\lib\ibmjsseprovider2.jar  
C:\myJVM\ sdk\jre\lib\ibmjaaslm.jar  
C:\myJVM\ sdk\jre\lib\ibmjaasactivelm.jar  
C:\myJVM\ sdk\jre\lib\ibmcertpathprovider.jar  
C:\myJVM\ sdk\jre\lib\server.jar  
C:\myJVM\ sdk\jre\lib\xml.jar
```

This output indicates that JVM 1 caused a classpath to be stored at address 0x2234FA6C in the cache and that it contains 17 entries, which are listed. If the classpath 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 classpath 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. It also indicates the classpath against which the class is stored and from which index in that classpath the class was loaded; in this case, the classpath is the same address as the one listed above. If a class is stale, it has !STALE! appended to the entry.

### URLs and Tokens

These are displayed in the same format as Classpaths. A URL is effectively the same as a Classpath, but with only one entry. A Token is in a similar format, but it is a meaningless String that can represent anything.

## Deploying Shared Classes

You cannot just "switch on" class sharing without considering how to deploy it sensibly for the chosen 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 will cause each user running the application to get a separate cache.

On UNIX platforms, if multiple users in the same operating system group are running the same application, use the **groupAccess** suboption, which will create 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.

If multiple applications or even different VM installs are to share classes, they can all share the same cache. Even different service releases of the 5.0 VM should be

able to share classes safely using the same cache. In general, the fewer caches created, the better. 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 clean up caches that aren't being used selectively.

On Windows, caches are stored as memory-mapped files in the user's directory in "Documents and Settings". Therefore, one user creating a cache "myCache" and another user creating a cache "myCache" will cause two different caches called "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. This behavior is different from that on UNIX, where only one "myCache" can exist and different users must have permissions to access it.

## Cache housekeeping

Unused caches on a system retain shared memory resources that could usefully be employed by another application. Therefore, ensuring that active caches are sensibly managed is important. The destroy and destroyAll utilities are used to explicitly destroy 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 a period of time. The suboption is added to the **-Xshareclasses** option and takes a parameter **<time>** in minutes. This option causes the VM 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. Therefore, a command line with **-Xshareclasses:name=myCache,expire=10000** automatically destroys any caches that have been unused for a week, before creating or connecting to myCache. Setting **expire=0** destroys all existing caches before initializing the shared classes support, so that a fresh cache is always created.

On UNIX platforms, the limitation with this housekeeping is that caches can be destroyed only by the user who created them or root. If the javasharedresources directory in /tmp is accidentally deleted, the control files that allow VMs to identify shared memory areas are lost and the shared memory areas are also therefore lost, although they still exist in memory. The next VM to start up after javasharedresources is deleted will therefore attempt to identify and destroy any such lost memory areas before recreating javasharedresources. Again, the VM can destroy only memory areas that were created by the user running that VM, unless it is running as root. (See the description of a hard reset in "3) Writing information in javasharedresources" on page 281.) Rebooting a system will cause all shared memory to be lost, although the control files still exist in javasharedresources. The existing control files will be deleted or reused and the shared memory areas recreated when the VM next starts up.

## Cache performance

Shared classes employs numerous optimizations to perform as well as possible under most circumstances. However, there are configurable factors which can affect shared classes performance, which are discussed here.

### Use of jar and zip files, not .class files

The cache keeps itself up-to-date with filesystem updates by constantly checking filesystem timestamps against the values in the cache. Because a classloader can obtain a lock on a jar file, after the jar has been opened and read, it is assumed

## Deploying Shared Classes

that the jar remains locked and does not need to be constantly checked. Because .class files can appear or disappear from a directory at any time, a directory in a classpath, particularly near the start, will inevitably have a performance impact on shared classes because it must be constantly checked for classes that might have appeared. For example, with 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 single class load. This checking also requires fabricating the expected directory from the class's package name. This operation can be expensive.

### Advantages of not filling the cache

A full shared classes cache is not a problem for any VMs connected to it. However, a full cache can place restrictions on how much sharing can be performed by other VMs or applications. Here is the rationale: 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 then another JVM loads the same class A from myOtherApp2.jar, only the one ROMClass will exist in the cache, with two pieces of metadata describing the two locations it came from. Therefore, if many classes are loaded by an application and the cache is 90% full, another install of the same application can use the same cache, and the amount of extra information that needs to be added about the second application's classes is minimal, even though they are separate copies on the file system. After the extra metadata has been added, both application installs can share the same classes from the same cache. However, if the first install had filled the cache completely, there would be no room for the extra metadata and the second application install would therefore not be able to share classes because it would not be able to update the cache. The same is true for classes that become stale and are redeemed. (See "Redeeming stale classes" on page 277). 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 classpath against which it was stored and the classpath of the caller classloader are "matched" to see whether the cache should return the class. The match does not have to be exact, but the result should be exactly the same as if the class were loaded from disk. Matching very long classpaths is initially expensive, but successful and failed matches are remembered, so that loading classes from the cache using very long classpaths is much faster than loading from disk.

### Growing classpaths

If at all possible, avoid gradually growing a classpath in a URLClassLoader using addURL(), because, 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 entry by entry using addURL(), you could 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 VMs. Any number of VMs can read from the cache at the same time as a single VM is writing to it. If many VMs start at the same time and no cache exists, one VM will win the race to create the cache and then all VMs will race to populate the cache with potentially the same classes. Multiple VMs concurrently loading the

same classes are coordinated to a certain extent by the cache itself to mitigate the effects of many VMs loading the same class from disk and racing to store it.

### Class GC with shared classes

Running with shared classes has no impact on class GC. Classloaders loading classes from the shared class cache can be garbage collected in exactly the same way as classloaders that load classes from disk. If a classloader is garbage collected, the ROMClasses it has added to the cache will persist.

---

## Dealing with runtime bytecode modification

Modifying bytecode at runtime is an increasingly popular way of engineering required function into classes. When a modification affects only a single JVM, such a modification is a safe thing to do; however, in a system in which classes are being shared between multiple JVMs, you must take care before caching modified bytecode. This section contains a brief summary of the tools that can help you to share modified bytecode.

### Potential problems with runtime bytecode modification

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. Even more unfortunately, if two JVMs start up at the same time using different modification agents, a mix of classes could be stored and both JVMs will probably throw VerifyErrors.
- Even if all JVMs sharing the same cache are using the same modification agents, the modifications being made must be predictable and repeatable, because, if the JVM loads a class from the shared class cache, it is read-only and cannot be modified, so all classes being loaded from the cache must already be modified correctly and completely.

In practice, modified bytecode can be shared safely if certain criteria are met:

- Modifications made are predictable and repeatable.
- 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. They are described in the next three sections. 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"

## Shared classes - dealing with runtime bytecode modification

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. The amount of checking involved, combined with the fact that bytecode must always be loaded from disk, inevitably has a performance impact on class loading. So, 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, JVM1 uses a JVMTI agent AGENT1, JVM2 uses no bytecode modification, JVM3 also uses AGENT1 and JVM4 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:

```
C:\JVM1\java -Xshareclasses:name=cache1,modified=myAgent1 myApp.ClassName  
C:\JVM2\java -Xshareclasses:name=cache1 myApp.ClassName  
C:\JVM3\java -Xshareclasses:name=cache1,modified=myAgent1 myApp.ClassName  
C:\JVM4\java -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 allows an application ClassLoader 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 classloader or bootstrap classloader does not support the use of partitions; instead, a SharedClassHelper must be used with a custom ClassLoader.

### Safemode

If you are experiencing odd or unexpected results from cached classes, or if you are experiencing VerifyErrors that you suspect might be caused by the wrong classes being returned or cached classes being incorrect, you can use a debugging mode called "safemode", which tells you whether the bytecode being loaded from the cache is correct for the JVM you are using.

**Safemode** is a suboption of **-Xshareclasses** and it does not "share classes". Use it with a populated cache (it will not add classes to a cache) and it forces the JVM to load all classes from disk and apply the modifications to those classes (if applicable). The ClassLoader then tries to store these classes that it has loaded in the cache. The class being stored is compared byte-for-byte against the class that would have been returned if the ClassLoader had not loaded the class from disk. If any bytes do not match, this mismatch is reported to stderr. Using safemode

guarantees that all classes are loaded from disk and provides a useful way of verifying whether the bytes being loaded from the shared class cache are the expected bytes.

Do not use Safemode in production because it is only a debugging tool and does not actually share classes.

### Further considerations for 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 within 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 would normally be required. However, it is useful to understand that the identical class is never stored in the cache more than once, even across modification contexts or partitions. Any number of metadata entries might describe the class and where it came from, but they will 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 (in the case where 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 from the cache classes that are out of date ("stale"). After a class has been marked stale, it is not returned by the cache if it is requested by a ClassLoader. Instead, the ClassLoader must reload the class from disk and store the updated version in the cache. This section goes into some detail about how the cache keeps up to date.

The cache manages itself to ensure that it deals with the following challenges:

- Jar and zip files are locked by ClassLoaders when they are in use, but can be updated when the JVM shuts down. Because the cache persists beyond the lifetime of any JVM using it, it checks for jar and zip updates.
- .class files (not in jar) can be updated at any time during the lifetime of a JVM. The cache checks for individual class file updates.
- .class files can appear or disappear in directories within classpaths at any time during the lifetime of a JVM. The cache checks the classpath for classes that have appeared or disappeared.
- .class files must be in a directory structure that reflects their package structure; therefore, when checking for updates, the correct directories must be searched.

## Shared classes - dynamic updates

Because class files contained in jars and zips and class files stored as .class files on the file system present different challenges, the cache treats these as two different types. Updates are managed by writing file system time stamps into the cache.

Note that ROMClasses 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 jar and zip files are time stamped and 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 updates to a file will cause an update to its directory.

If a zip or jar file does not exist, the classpath containing it can still be added to the cache, but ROMClasses from this entry are not stored. If a ROMClass is being added to the cache from a directory and it does not exist as a .class file, it is not stored.

Time stamps can also be used to determine whether a ROMClass being added is a duplicate of one that already exists in the cache.

If a classpath entry is updated on the file system and this entry is out of sync with a classpath time stamp in the cache, the classpath is added again and time stamped again in its entirety. Therefore, when a ROMClass is being added to the cache and the cache is searched for the caller's classpath, any potential classpath matches are also time stamp-checked to ensure that they are up-to-date before the classpath is returned.

## Finding classes

When finding a class in the cache, more checks must be made than when storing classes.

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 jar and zip entry in the caller's 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 could mean that another version of the class being returned might have been added earlier in the classpath.

Additionally, any classpath entries that are directories might contain .class files that will "shadow" the potential match that has been found. Class files might appear or disappear in these directories at any point. Therefore, when the classpath is walked and jars and zips are checked, directory entries are also checked to see whether any .class files have appeared unexpectedly. This check involves building a string out of the classpath entry, the package names, and the class name, and then looking for the classfile. This procedure is expensive if many directories are being used in classpaths. Therefore, using jar files gives better shared classes performance.

## Marking classes as stale

When a jar or zip classpath entry is updated, all of the classes in the cache that could potentially have been affected by that update are marked "stale". When an individual .class file is updated, only the class or classes stored from that .class file are marked stale. The stale marking used is pessimistic because the cache does not know the contents of individual jars and zips.

For example, therefore, for the following classpaths where c has become stale:

- a;b;c;d** c could now contain new versions of classes in d; therefore, classes in both c and d are all stale.
- c;d;a** c could now contain new versions of classes in d and a; therefore, classes in c, d, and a are all stale.

So, classes in the cache that have been loaded from c, d, and a are marked stale. Therefore, it takes only a single update to one jar file to potentially cause many classes in the cache to be marked stale. To ensure that there is not massive duplication as classes are unnecessarily restored, stale classes can be "redeemed" if it is proved that they are not in fact stale.

## Redeeming stale classes

Because classes are pessimistically marked stale when an update occurs, possibly many or most of the classes marked stale have not in fact been updated. So, when a ClassLoader stores a class that effectively "updates" a stale class, you can "redeem" the stale class if you can prove that it has not in fact changed.

For example, class X is stored from c with classpath a;b;c;d. Suppose that a is updated, meaning that a could now contain a new version of X (it actually does not) but all classes loaded from b, c, and d are marked stale. Another JVM wants to load X, so it asks the cache for it, but it is stale, so the cache does not return the class. The ClassLoader therefore loads it from disk and stores it, 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 classloader internally in the JVM, but any other Java ClassLoader 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 that enable Java ClassLoaders to exploit the shared classes features in the JVM. The `java.net.URLClassLoader` shipped with the SDK has been modified to use a `SharedClassURLClasspathHelper` and any ClassLoaders that extend `java.net.URLClassLoader` inherit this behavior. Custom ClassLoaders that do not extend `URLClassLoader` but want to share classes must use the Java Helper API, so this section 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 for these classes is shipped with the SDK (some of which is reproduced here).

### `com.ibm.oti.shared.Shared`

The Shared class contains two important static functions: `getSharedClassHelperFactory()` and `isSharingEnabled()`. If `-Xshareclasses` is specified on the command line and sharing has been successfully

## Shared classes - using the java Helper API

initialized, isSharingEnabled() returns true and getSharedClassHelperFactory() will return a com.ibm.oti.shared.SharedClassHelperFactory, which is a singleton factory that manages the Helper APIs. To get a Helper API, you need to get a SharedClassHelperFactory.

### com.ibm.oti.shared.SharedClassHelperFactory

SharedClassHelperFactory provides an interface used to create various types of SharedClassHelper for ClassLoaders. ClassLoaders and SharedClassHelpers have a one-to-one relationship. Any attempts to get a helper for a ClassLoader that already has a different type of helper results in a HelperAlreadyDefinedException.

Because ClassLoaders and SharedClassHelpers have a one-to-one relationship, calling findHelperForClassLoader() returns a Helper for a given ClassLoader 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 ClassLoader. Typically used by ClassLoaders, which require complete control over cache contents.
- **SharedClassURLHelper.** Store and find classes using a file system location represented as a URL. For use by ClassLoaders, which 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 ClassLoaders that load classes using a URL classpath

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 also that classes stored using the URL Helpers are updated dynamically by the cache (see “Understanding dynamic updates” on page 275) but classes stored by the SharedClassTokenHelper are not updated by the cache because the Tokens are meaningless Strings, so it has no way of obtaining versioning information. For a detailed description of each helper and how to use it, refer to the javadoc 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.

## General API Helper usage

Regardless of the Helper API being used, they all provide two key functions:

### findSharedClass

Called after the ClassLoader 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[]), this class should be passed to defineClass(), which will define the class for that JVM and return it as a java.lang.Class object. Note that the byte[] returned by findSharedClass is not the actual class bytes, so it cannot be instrumented or manipulated in the same way as class bytes loaded off 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 `ClassLoader` 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`.

Because only classes can be stored in the cache, you must resolve how to deal with metadata that cannot be stored. An example of this is `java.security.CodeSource` or `java.util.jar.Manifest` objects that are derived from Jar files. The recommended way to deal with this is, for each jar, always load the first class from the jar regardless of whether it exists in the cache or not. This load initializes the required metadata in the `ClassLoader`, which can then be cached internally. When a class is then returned by `findSharedClass`, the function indicates from where the class would have been loaded, and that means that the correct cached metadata for that class can then 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 it will still be shared. However, although it is handled correctly, a `ClassLoader` that uses only `storeSharedClass` is less efficient than one that also makes proper use of `findSharedClass`.

---

## 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 very 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 “Running with method trace” on page 221 for details. Five levels of trace are provided, level 1 giving essential initialization and runtime information, up to level 5, which is very detailed.

Shared classes trace output does not include trace from the port layer functions that deal with shared memory and shared semaphores. It also does not include trace from the Helper API natives. Port layer trace is enabled using the `j9prt` trace component and trace for the Helper API natives is enabled using the `j9jcl` trace component.

### Why classes in the cache might not be found or stored

To help you diagnose why classes might not be being found or stored in the cache as expected, here is a quick guide. (F) indicates a class not being found and (S) a class not being stored.

#### The class is stale (F)

As explained in “Understanding dynamic updates” on page 275, if a class has been marked as “stale”, it is not returned by the cache.

## Debugging problems with shared classes

### The Classpath entry being used is not yet confirmed by the SharedClassURLClasspathHelper (F)

Classpath entries in the SharedClassURLClasspathHelper must be "confirmed" before classes can be found for these entries. A classpath entry is confirmed by having a class stored for that entry. For more information about confirmed entries, refer to the SharedClassHelper javadoc.

### The class being stored does not exist on the filesystem (S)

The class might have been generated or might come from a URL location that is not a file.

### Safemode is being used (FS)

Classes are not found or stored in the cache in safemode. This behavior is expected for shared classes. See "Safemode" on page 274.

### The cache is corrupt (FS)

In the unlikely event that the cache is corrupt, no classes can be found or stored.

### A SecurityManager is being used and the permissions have not been granted to the ClassLoader (FS)

SharedClassPermissions need to be granted to application ClassLoaders so that they can share classes with a SecurityManager. For more information, see the *SDK and Runtime User Guide* for your platform.

## Dealing with initialization problems

Shared classes initialization requires the following operations to succeed:

1. Create a shared memory area.
2. Create a shared semaphore.
3. Write information about (1) and (2) in a temporary directory.

Although this sequence might sound straightforward, the operating system might refuse to perform any one of these three operations for a number of reasons. Because a failure could have many potential reasons, it is difficult to provide detailed information on the command line following an initialization failure, so the following sections describe some of the common reasons for failure. If you cannot determine the reason for initialization 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, so only a brief summary of potential reasons for failure is provided here.

### 1) Creating a shared memory area

Different operating systems have different restrictions on creating a shared memory segment:

#### 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 available disk space and file write permissions.

**UNIX** The **SHMMAX** operating system environment variable by default is set quite low. **SHMMAX** limits the size of shared memory segment that can be allocated. If a cache size greater than **SHMMAX** is requested, the VM attempts to allocate SHMMAX and will output a message indicating that **SHMMAX** should be increased. For this reason, the default cache size is currently only 16 MB.

**z/OS** Before using shared classes on z/OS, it is important to check in the *z/OS SDK and Runtime Environment User Guide* for details of APARs that must be installed on the system. Also, check the operating system environment variables, as detailed in the user guide. (Note that, on z/OS, the requested cache sizes are deliberately rounded to the nearest megabyte.)

### 2) Creating a shared semaphore

This operation has fewer causes for failure. To create the semaphore, it is necessary to be able to write to the javasharedresources directory, so write permissions are needed.

### 3) Writing information in javasharedresources

On all platforms, data must be written into a "javasharedresources" directory, which must be created by the first JVM that needs it. On UNIX, this directory is in /tmp/javasharedresources and on Windows it is in C:\Documents and Settings\<username>\Local Settings\Application Data. On Windows, the memory-mapped file is written here. On UNIX this directory is used only to store small amounts of metadata that identify the semaphore and shared memory areas.

Problems writing to this directory are the most likely cause of initialization failure. The default cache name is created with the username incorporated to prevent clashes if different users try to share the same default cache, but all shared classes users must have permissions to write to javasharedresources. Furthermore, the user running the first JVM to use shared classes on a system must have permission to create the javasharedresources directory.

Regarding permissions, by default on UNIX caches are created with user-only access, so two users cannot share the same cache unless the "groupAccess" command-line option is explicitly 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 will occur, because user B does not have permissions to access the existing cache called myCache. Also, caches can be destroyed only by the user who created them (or root), even if groupAccess is used.

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 UNIX system, it is advisable to run this command as root.)
2. Delete the javasharedresources directory and all of its contents. For Windows, that is all you have to do.
3. On UNIX, if initialization problems have been experienced, possibly all the memory areas and semaphores created by the JVM will not have been removed by step 1. This problem is addressed the next time you start the VM. If the VM starts without javasharedresources, an automated cleanup is triggered when the VM initializes and it looks for any remaining shared memory areas that are shared class caches and attempts to destroy them. Because only the creator of a shared memory area or root can destroy a cache, after deleting javasharedresources you are advised to run the JVM with **-Xshareclasses** as root and thus ensure that the system is completely reset. The VM will then automatically recreate javasharedresources after the cleanup.

## Dealing with verification problems

Verification problems (which mostly appear as `java.lang.VerifyErrors`) are potentially caused by the cache returning incorrect class bytes. This problem should not occur under normal usage, but there are two situations in which it could happen:

## Debugging problems with shared classes

- 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 “Safemode” on page 274) 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 output 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 output 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

It is important to understand that the cache can be set only when the cache is created because it is a fixed size. 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 adaptor has been written specifically for use with OSGi, which allows OSGi ClassLoaders to access the class cache.

---

## Chapter 33. Tracing Java applications and the JVM

JVM Trace is a low-overhead trace facility that is provided in all IBM-supplied JVMs. In most cases, the trace data is kept in compact binary format, with variable-length trace records from 8 to 64 KB. A cross-platform Java formatter is supplied to format the binary trace files. Tracing is enabled by default, together with a small set of trace points going to in-storage 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
- “Default Memory Management tracing” on page 284
- “Where does the data go?” on page 285
- “Controlling the trace” on page 286
- “Determining the tracepoint ID of a tracepoint” on page 299
- “Application trace” on page 300

The trace tool provides an extremely powerful ability to diagnose the JVM.

---

### What can be traced?

What can be traced depends on:

- Tracing methods
- Tracing applications
- Internal trace

### Tracing methods

You can trace entry to and exit from methods for selected classes. Using the **methods** trace option, you can select method trace by class, method name, or both. Wildcards can be used, and a "not" operator allows for complex selection criteria. Note that this option selects only the methods that are to be traced. The MT trace component must be selected for a given trace destination. For example:

```
-Xtrace:methods={*.*,!java/lang/*.*},print=mt
```

This command routes method trace to stderr for all methods and for all classes except those that start with java/lang.

### Tracing 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. API in the com.ibm.jvm.Trace class is provided to register a Java application for trace and later to make trace entries. You can control the tracepoints at startup or enable them dynamically by using Java or C API. When trace is not enabled, little overhead is caused. Note that an instrumented Java application runs only on an IBM-supplied JVM.

## What can be traced?

### Internal trace

The IBM Virtual Machine for Java is extensively instrumented for trace, as described in this chapter. Interpretation of this trace data requires knowledge of the internal operation of the JVM, and is provided for support personnel who diagnose JVM problems.

No guarantee is given that tracepoints will not vary from release to release and from platform to platform.

---

## Default tracing

JVM initialization starts trace with a small set of trace points that will be captured to wrap around in-storage buffers. It uses the equivalent of the following command-line option:

-Xtrace:maximal=all{level1},exception=j9mm(gclogger)

You can then find these trace points in a system dump (extracted using jdmpview) or snap trace, available for diagnostics on many of the default error conditions. Chapter 23, “Using dump agents,” on page 195 describes these conditions.

If you specify **-Xtrace** on the command line, or bring it in from a properties file, the set of active trace points is cleared.

---

## Default Memory Management tracing

The **exception=j9mm{gclogger}** clause of the default trace set ensures that a history of garbage collection cycles that have occurred in the vm are recorded in any snap dumps that may be taken. 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 so that they are not overwritten by the higher frequency tracepoints of the vm.

This mechanism ensures that any snap dumps or similar always contain a record of the most recent memory management history, regardless of how much vm activity has occurred since the garbage collection cycle was last invoked.

As long as at least one garbage collection cycle has occurred in a traced vm run, any snap files created should contain a garbage collection history similar to the following after it has been formatted:

```
08:11:07.137116731 00173700 j9mm.50 Event SystemGC start: exclusiveaccessms=0.017  
    newspace=0/0 oldspace=3539000/4194304 loa=209408/209408  
08:11:07.137144246 00173700 j9mm.52 Event GlobalGC start: weakrefs=80 soft=8  
    phantom=0 finalizers=16 globalcount=1 scavengecount=0  
08:11:07.137156475 00173700 j9mm.54 Event Mark start  
08:11:07.138062256 00173700 j9mm.55 Event Mark end  
08:11:07.138066147 00173700 j9mm.56 Event Sweep start  
08:11:07.138130350 00173700 j9mm.57 Event Sweep end  
08:11:07.138134241 00173700 j9mm.60 Event Class unloading start  
08:11:07.138138132 00173700 j9mm.61 Event Class unloading end  
08:11:07.138157031 00173700 j9mm.53 Event GlobalGC end: workstackoverflow=0  
    overflowcount=0 weakrefs=78 soft=0 phantom=0 finalizers=14 newspace=0/0  
    oldspace=3727248/4194304 loa=209408/209408  
08:11:07.138165369 00173700 j9mm.51 Event SystemGC end: newspace=0/0  
    oldspace=3727248/4194304 loa=209408/209408
```

## Where does the data go?

Trace data can go into:

- In-storage 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
- A combination of the above

### Placing trace data into in-storage buffers

The use of in-storage buffers for trace is a very efficient method of running trace because no explicit I/O is performed until either a problem is detected, or an API is used to snap the buffers to a file. Buffers are allocated on a per-thread principle. This principle removes contention between threads and prevents trace data for individual threads from being swamped by 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 `-Xtrace:buffers=<size>` to control the size of the buffer that is allocated to each thread.

**Note:** On some computers, power management affects the timers that trace uses, and gives misleading information. This problem affects mainly Intel(TM)-based mobiles, but it can occur on other architectures. For reliable timing information, disable power management.

To examine the trace data, you must snap or dump, then format the buffers.

### Snapping buffers

Buffers are snapped when:

- An uncaught Java exception 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 file is placed into the current working directory with a name of the format `Snapnnnn.yyyymmdd.hhmmss.th.process.trc`, where `nnnn` is a sequence number starting at 0001 (at JVM startup), `yyyymmdd` is the current date, `hhmmss` is the current time, and `process` is the process identifier.

### Dumping buffers

You can also dump the buffers by using the operating system dump services. You can then extract the buffers from the dump by using the Dump Viewer.

### Placing trace data into 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. 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

## Where does the data go?

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. See the **output** option for details. You must use the trace formatter to format trace data from the file.

**Note:** Because of the buffering of trace data, if the normal JVM termination is not performed, residual trace buffers might not be flushed to the file. Snap traces do not occur, and the trace bytes are not flushed except when a fatal operating-system signal is received. The buffers can, however, be extracted from a system dump if that is available.

## External tracing

You can route trace to an agent by using JVMRI TraceRegister. This allows a callback routine to be invoked when any of the selected tracepoints is found in real time; that is, no buffering is done. The trace data is in raw binary form.

## Tracing to stderr

For lower volume or non-performance-critical tracing, the trace data can be formatted and routed to stderr in real time. See Chapter 27, “Using method trace,” on page 221.

## Trace combinations

Most trace destinations can be combined, with the same or different trace data going to different destinations. The exception to this is in-storage trace and trace to a file, which are mutually exclusive.

---

## Controlling the trace

You can control the trace in several ways:

- By using trace options when launching the JVM
- By using a trace properties file
- By dynamically using Java API
- By using trace trigger events
- By using the C API from inside the JVM
- From an external agent, by using JVMRI

### Notes:

1. By default, trace options equivalent to the following are enabled:

```
-Xtrace:maximal=all{level1},maximal=j9mm{gclogger}
```

2. Whenever the JVM is run, it uses **IBM\_JAVA\_OPTIONS** if set.

**IBM\_JAVA\_OPTIONS** includes any Java utilities, such as the trace formatter, the dump extractor, and the dump formatter. If the JVM uses **IBM\_JAVA\_OPTIONS**, unwanted effects or loss of diagnostic data can occur. For example, if you are Using **IBM\_JAVA\_OPTIONS** to trace to a file, that file might be overwritten when the trace formatter is called. To avoid this problem, add %d, %p, or %t into the filename to make it unique. Go to “Detailed descriptions of trace options” on page 289 and see the appropriate trace option description for more information.

## Specifying trace options

The primary way to control trace is through trace options that you specify either by using the **-Xtrace** option on the launcher command-line or the **IBM\_JAVA\_OPTIONS** environment variable. Some trace options have the form

<name>, while others are of the form <name>=<value>, where <name> is case-sensitive. Except where stated, <value> is case insensitive; the exceptions to this rule are filenames 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 that this only applies to options specified on the command-line - not those 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
```

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>
```

Users of UNIX style shells should be aware that unwanted shell expansion might occur because of the characters used in the trace options. To avoid unpredictable results, you should enclose this command line option in quotes. For example:

```
java "-Xtrace:<name>,<another_name>=<value>" HelloWorld
```

For more information, please see the manual for your shell.

## Trace options summary

This section describes:

- “Options that control tracepoint selection”
- “Options that indirectly affect tracepoint selection” on page 288
- “Triggering and suspend or resume” on page 288
- “Options that specify output files” on page 288
- “MiscellaneousTrace control options” on page 288

### Options that control tracepoint selection

These options enable and disable tracepoints. They also determine the destination for 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 in-core buffers. If you are going to send the data to a file, you must use an output option to specify the destination filename.

These properties have equivalents in the Java and JVMRI API that was mentioned earlier.

*Table 13. Options that control tracepoint selection*

<b>minimal</b>	Trace selected tracepoints (identifier and timestamp only) to in-core buffer. Associated trace data is not recorded.
<b>maximal</b>	Trace selected tracepoints (identifier and timestamp and associated data) to in-core buffer.
<b>count</b>	Count the number of times selected tracepoints are called in the life of the JVM.

## Controlling the trace

*Table 13. Options that control tracepoint selection (continued)*

<b>print</b>	Trace selected tracepoints to stderr with no indentation.
<b>iprint</b>	Trace selected tracepoints to stderr with indentation.
<b>external</b>	Route selected tracepoints to a JVMRI listener.
<b>exception</b>	Trace selected tracepoints to an in-core buffer reserved for exceptions.

### Options that indirectly affect tracepoint selection

These options affect the availability of particular tracepoints but unless you specify them with a tracepoint selection option, they have no effect other than possibly degraded performance.

*Table 14. Options that indirectly affect tracepoint selection*

<b>methods</b>	Select classes and methods to trace.
----------------	--------------------------------------

### Triggering and suspend or resume

These trace options provide mechanisms to tailor trace and trigger actions at specified times

*Table 15. Triggering and suspend or resume*

<b>trigger</b>	Trigger events by tracepoint, group or method entry/exit.
<b>suspend</b>	Suspend tracepoints globally (for all threads).
<b>resume</b>	Resume tracepoints globally (not really useful, but here for completeness).
<b>suspendcount</b>	Initial thread suspend count.
<b>resumecount</b>	Initial thread resume count.

### Options that specify output files

These options determine whether trace data is directed to a file. For the first two options, you must activate tracepoints by using a tracepoint selection options or through the various API that were mentioned earlier.

*Table 16. Options that specify output files*

<b>output</b>	Select output file name and options for trace data from tracepoints that were selected through the minimal and maximal properties.
<b>exception.output</b>	Select output file name and options for trace data from tracepoints that were selected through the exception property.

### Miscellaneous Trace control options

*Table 17. Miscellaneous Trace control options*

<b>properties</b>	Specify a file containing options for trace.
-------------------	--

Table 17. Miscellaneous Trace control options (continued)

<b>buffers</b>	Modify buffer size and allocation.
----------------	------------------------------------

## Detailed descriptions of trace options

The options are processed in the sequence in which they are described here.

### **properties[=properties\_filespec]**

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/EBCDIC file that contains trace options. If **properties\_filespec** 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 appear 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.

**Note:** 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**.

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:\trcprops\gc:

```
-Xtrace:properties=c:\trcprops\gc
```

Here is an example property file:

```
minimal=all
// maximal=j9mm
maximal=j9shr
buffers=20k
output=c:\traces\classloader.trc
print=tnid(j9vm.23-25)
```

### **buffers=nnnk | nnnm[,dynamic | nodynamic]**

This option specifies the size of the buffer as nnn KB or MB. This buffer is allocated for each thread that makes trace entries. If external trace is enabled, this value 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**

## Controlling the trace

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.

**Important:** 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 8 KB buffers:

```
-Xtrace:buffers=8k
```

or in a properties file:

```
buffers=8k
```

Trace buffers 2 MB per thread:

```
-Xtrace:buffers=2m
```

or in a properties file:

```
buffers=2m
```

Trace to only two buffers per thread, each of 128 KB:

```
-Xtrace:buffers={128k,nodynamic}
```

or in a properties file:

```
buffers=128k,nodynamic
```

**minimal[=[!!]tracepoint\_specification[...]]**,  
**maximal[=[!!]tracepoint\_specification[...]]**, **count[=[!!]tracepoint\_specification[...]]**,  
**print[=[!!]tracepoint\_specification[...]]**, **iprint[=[!!]tracepoint\_specification[...]]**,  
**exception[=[!!]tracepoint\_specification[...]]**,  
**external[=[!!]tracepoint\_specification[...]]**

### Summary

These options control which individual tracepoints are activated at runtime and the implicit destination of the trace data. **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.

Tracepoints that are activated with **count** are only counted. The totals are written to a text file called dgTrcCounters in the current directory at JVM termination.

Tracepoints that are activated with **print** or **iprint** are routed to stderr.

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 system property.

**External** trace data is passed to a registered trace listener. Note that all these properties are independent of each other and can be mixed and matched in any way that you choose.

Multiple statements of each type of trace are allowed and their effect is cumulative. Of course, you would have to use a trace properties file for multiple trace options of the same name.

### Types of trace

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**.

The **count** option requests that 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}
```

The **print** option causes the specified tracepoints to be routed to `stderr` in real-time. The tracepoints are formatted by `J9TraceFormat.dat`, which must be available at runtime. `TraceFormat.dat` is shipped in `sdk/jre/lib` and is automatically found by the runtime.

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 selection."

**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.)

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.

### Tracepoint selection:

If no qualifier parameters are entered, all tracepoints are enabled, except for **exception** trace, where the default is **all {exception}**.

The **tracepoint\_specification** is as follows:

## Controlling the trace

- `[!]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 one of:
  - all
  - The JVM subcomponent (that is, dg, j9trc, j9vm, j9mm, j9bcu, j9vrb, java.awt, awt\_dnd\_datatransfer, Audio, mt, fontmanager, net, awt\_java2d, awt\_print, or nio)
- 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 would 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. This constitutes the component name of the tracepoint, followed by its integer number within that component. For example, j9mm.49, j9shr.20-29, j9vm.15, etc.

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 will be included.

### 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!=tpnid{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`

All exception tracepoints:

`-Xtrace:exception=all{exception}`

All exception tracepoints in j9bcu:

```
-Xtrace:exception=j9bcu
Tracepoints j9prt.15 and j9shr.12:
-Xtrace:exception=tpnid{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 very important and 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 TPID keyword.

The default is level 9.

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 below are enabled for all components. In the second example, all level 1 tracepoints are enabled, as well as all level2 tracepoints in j9mm, and all tracepoints up to level 9 are enabled in j9bcu. Note that the level applies only to the current component, so if multiple trace selection components appear 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 above for the j9mm component. For example:

```
-Xtrace:print={all,!j9trc(15),!j9mm(16)}
```

enables trace for all components at level 9 (the default), but disables level 6 and above for the locking component, and level 7 and above for the storage component.

### Examples:

Count all level zero and one tracepoints hit:

```
-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}}
```

### methods=method\_specification[...]

This trace option identifies which classes and methods are to be prepared to be traced. You can then trace these methods by selecting the MT component

## Controlling the trace

though the normal trace selection mechanism. When more than one specification is made, it is cumulative, as if processed from left to right. Although method trace works with the JIT on, input parameters cannot be traced if the JIT is active.

**Important:** This trace option selects only the methods that are to be traced.

You must use one of the trace selection properties to select the tracepoints that are in the MT component.

The **method\_specification** is:

- `[!][*]class[*][.*]method[*]][0]`, where

<code>!</code>	is a logical not. That is, the class or methods that are specified immediately following the <code>!</code> are deselected for method trace.
<code>*</code>	is a wildcard that can appear at the beginning, end, or both, of the class and method names.
<code>class</code>	is the package or class name. Note that the delimiter between parts of the package name is a forward slash, <code>'/'</code> , even on platforms like Windows that use a backward slash as a path delimiter.
<code>.</code>	is the delimiter between the class and method.
<code>method</code>	is the method name.

### Examples:

Select all methods for all classes and print them with indentation:

`-Xtrace:methods=*,iprint=mt`

All methods that are in `java/lang/String`. The trace data will be placed in internal buffers:

`-Xtrace:methods=java/lang/String.*,maximal=mt`

All methods that contain a "y" in classes that start with `com/ibm` and print them:

`-Xtrace:methods=com/ibm*.y*,print=mt`

All methods that contain a "y" and do not start with an "n" in classes that start with `com/ibm` and print them:

`-Xtrace:methods={com/ibm*.y*,!n*},iprint=mt`

### **output=trace\_filespec[,nnnm[generations]]**

This trace options indicates that minimal, or maximal trace data, or both, must be sent to `trace_filespec`. 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 until all disk space has been used.
- 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 `trace_filespec`.
  - To include the pidnumber of the process that is generating the tracefile, specify "%p" as part of the `trace_filespec`.
  - To include the time (in 24-hour hhmmss format) in the trace filename, specify "%t" as part of the `trace_filespec`.

- 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. Therefore, if  $x$  generations of  $n$  MB files are specified, the worst case is that only  $((x - 1) * n \div x)$  MB of trace data might be available. 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
```

Output goes to trace and will wrap at 2 MB:

```
-Xtrace:output={trace,2m}
```

Output goes to gc0.trc, gc1.trc, gc2.trc, each 10 MB in size:

```
-Xtrace:output={gc#.trc,10m,3}
```

Output filename contains today's date in yyyyymmdd format (for example, traceout.20041025.trc):

```
-Xtrace:output=traceout.%d.trc
```

Output file contains the number of the process (the PID number) that generated it (for example, tracefrompid2112.trc):

```
-Xtrace:output=tracefrompid%p.trc
```

Output filename contains the time in hhmmss format (for example, traceout.080312.trc):

```
-Xtrace:output=traceout.%t.trc
```

#### **exception.output=exception\_trace\_filespec[,nnnm]**

This trace option indicates that **exception** trace data should be directed to **exception\_trace\_filespec**. 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 until all disk space has been used.

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 "yyyyymmdd" format) in the trace filename, specify "%d" as part of the **exception\_trace\_filespec**.
- To include the pidnumber of the process that is generating the tracefile, specify "%p" as part of the **exception\_trace\_filespec**.
- To include the time (in 24-hour hhmmss format) in the trace filename, specify "%t" as part of the **exception\_trace\_filespec**.

## Controlling the trace

### Examples:

Trace output goes to /u/traces/exception.trc. No size limit:  
-Xtrace:exception.output=/u/traces/exception.trc

Output goes to except and wraps at 2 MB:  
-Xtrace:exception.output={except,2m}

Output filename contains today's date in yyyyymmdd format (for example, traceout.20041025.trc):  
-Xtrace:exception.output=traceout.%d.trc

Output file contains the number of the process (the PID number) that generated it (for example, tracefrompid2112.trc):  
-Xtrace:exception.output=tracefrompid%p.trc

Output filename contains the time in hhmmss format (for example, traceout.080312.trc):  
-Xtrace:exception.output=traceout.%t.trc

### suspend

Suspends tracing globally (for all threads and all forms of tracing) but leaves tracepoints activated.

### Example:

Tracing suspended:  
-Xtrace:suspend

### 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

### suspendcount=<count>

This trace option is for use with the **trigger** option (see "trigger" on page 297).

This **suspendcount=<count>** 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.

### Example:

Start with all tracing turned on. Each thread stops tracing when it has had three suspendthis actions performed on it:

-Xtrace:suspendcount=3

### resumecount=count

This system property is for use with the **trigger** property (see "trigger" on page 297).

This **resumecount=<count>** system property 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> resume this action before it starts tracing.

**Note:** You cannot use resumecount and suspendcount together because they both set the same internal counter.

**Example:**

Start with all tracing turned off. Each thread starts tracing when it has had three resumethis actions performed on it:

```
-Xtrace:resumecount=3
```

**trigger=clause[,clause][,clause]...**

This trace option determines when various triggered trace actions should occur. Supported actions include turning tracing on and off for all threads, turning tracing on or off for the current thread, or producing a variety of dumps.

**Note:** This trace option does not control what is traced. It controls only whether what has been selected by the other trace options is produced as normal or is blocked.

Each clause of the **trigger** option can be **tpid{...}**, **method{...}**, **group{...}**, or **threshold{}**. You can specify multiple clauses of the same type if required, but you do not need to specify all types. The clause types are:

**Method{methodspecl,entryAction],[exitAction][,delayCount][,matchcount]}**

On entering a method that matches **methodspecl**, perform the specified **entryAction**. On leaving it, perform the specified **exitAction**. If you specify a **delayCount**, the actions are performed only after a matching **methodspecl** has been entered that many times. If you specify a **matchCount**, **entryAction**, and **exitAction** will be 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**, perform the specified action. 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** will be performed at most that many times.

**tpid{tpid | tpidRange,action[,delayCount][,matchcount]}**

On finding the specified active **tpnid** (tracepoint id) or a **tpnid** that falls inside the specified **tpnidRange**, perform the specified action. 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** will be performed at most that many times.

**Actions:**

Wherever an action must be specified, you must select from the following choices:

**suspend**

Suspend ALL tracing (except for special trace points).

**resume**

Resume ALL tracing (except for threads that are suspended by the action of the resumecount property and Trace.suspendThis() calls).

**suspendthis**

Increment the suspend count for this thread. If the suspend-count is greater than zero, all tracing for this thread is prevented.

## Controlling the trace

### resumethis

Decrement the suspend count for this thread. If the suspend-count is zero or below, tracing for this thread is resumed.

### coredump (or sysdmp)

Produce a coredump.

### javadump

Produce a javadump or javacore.

### heapdump

Produce a heap dump (see Chapter 25, “Using Heapdump,” on page 213).

### snap

Snap all active trace buffers to a file in the current working directory. The name of the file is in the format Snapnnnn.yyyymmdd.hhmmss.thpppp.trc, where nnnn is the sequence number of the snap file since JVM startup, yyyymmdd is the date, hhmmss is the time, and thpppp is the process id in decimal with leading zeroes removed.

### abort

Halt the execution of the JVM.

### segv

Cause a segmentation violation. (Intended for use in debugging.)

### Examples:

- Start tracing this thread when it enters any method in java/lang/String and stop tracing when it leaves it:  
`-Xtrace:resumecount=1  
-Xtrace:trigger={method{java/lang/String.*,resumethis,suspendthis}}`
- Resume all tracing when any thread enters a method in any class that starts with “error”:  
`-Xtrace:trigger={method{*.error*,resume}}`
- When you reach the 1000th and 1001st tracepoint from the “jvmri” trace group, produce a core dump.

**Note:** Without matchcount there would be a risk of 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 23, “Using dump agents,” on page 195 for more information.

- Trace (all threads) while my application is active only; that is, not startup or shutdown. (The application name is “HelloWorld”):  
`-Xtrace:suspend,trigger={method{HelloWorld.main,resume,suspend}}`

## Using the trace formatter

The trace formatter is a Java program that runs on any platform and can format a trace file from any platform. The formatter, which is shipped with the SDK in core.jar, also requires a file called TraceFormat.dat, which contains the formatting templates. This file is shipped in jre/lib.

### Invoking the trace formatter

Type:

`java com.ibm.jvm.format.TraceFormat input_filespec [output_filespec] [options]`

where `com.ibm.jvm.format.TraceFormat` is the traceformatter class, `input_filespec` is the name of the binary trace file to be formatted, `output_filespec` is the optional output filename. If it is not specified, the default output file name is `input_filespec.fmt`.

The options are:

- `-summary` specifies that a summary of the trace file is printed.
- `-dat` gives the location of the `.dat` files for processing a trace files produced by a pre 5.0 vm. The dat files must be from a vm with a greater or equal version number to the trace files. The dat files can not be from a 5.0 vm.
- `-uservmid` specifies a user string to be inserted in each formatted tracepoint. The string aids reading or parsing when several different vms or vm runs are traced for comparison. It allows easy identification within the file of which vm owns the trace output that is being inspected.
- `-indent` specifies that the formatter will indent trace messages at each Entry trace point and outdent trace messages at each Exit trace point. The default is not to indent the messages.

Examples of formatting binary trace file `trace1`:

- Produce a summary of the trace file:  
`java com.ibm.jvm.format.TraceFormat trace1 -summary`

## Trace properties file

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. You can remove unwanted tracepoints by using the `!TPNID(xxxxxx)` parameter.

## What to trace

JVM trace can produce large amounts of data in a very short time. Before running trace, think carefully about what information you need to solve the problem. In many cases, where you need only the trace information that is produced shortly before the problem occurs, consider using the `wrap` option. In many cases, just use internal trace with an increased buffer size and snap the trace when the problem occurs. If the problem results in a thread stack dump or operating system signal or exception, trace buffers are snapped automatically to a file that is in the current directory. The file is called: `Snapnnnn.yyyymmdd.hhmmssth.process.trc`.

You must also think carefully about which components need to be traced and what level of tracing is required. For example, if you are tracing a suspected garbage collection problem, it might be enough to trace all components at level 1 or 3, and `j9shr` at level 9, while `maximal` can be used to show parameters and other information for the failing component.

## 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.

## Controlling the trace

These tracepoints are also recorded within 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. Below is a 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 (5.0 in the case above). 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.

So, for example, if you find that a problem occurred somewhere close to the issue of Trc\_BCU\_VMInitStages\_Event, you would rerun the application with **-Xtrace:print=tpid{j9bcu.0}**. That command would 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 tpid 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

```
int registerApplication(String application_name, String[] format_template)
```

Use the `registerApplication()` method to specify the application to register with application trace. The `application_name` argument is the name of the application

you want to trace; application\_name must be the same as the one that you specify at JVM startup – that is, the application that you want to trace. The format\_template argument is an array of printf-like format strings. 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 runtime. The first character of each template identifies the type of tracepoint (entry, exit, event, exception or exception exit) followed by a blank, followed by the format string. The trace types are defined as statics in 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 that you must use on further trace() calls. If the registration call fails for any reason, it returns -1.

## Tracepoints

The following 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);
```

## Controlling the trace

```
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.

### Example HelloWorld with application trace

The code below illustrates a “HelloWorld” application with application trace:

```
import com.ibm.jvm.Trace;
public class HelloWorld
{
    static int handle;
    static String[] templates;
    public static void main (String[] args)
    {
        templates = new String[8];
        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 requested on 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 runtime

At runtime, you can enable one or more applications for application trace. For example, in the case of the “HelloWorld” application described above:

```
java HelloWorld iprint=HelloWorld
```

The HelloWorld example uses the Trace.set() API to pass any arguments to trace, enabling all of the HelloWorld tracepoints to be routed to stderr. Invoking the HelloWorld application in this way outputs:

```

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 obtain a similar result by specifying iprint on the command line:

```
java -Xtrace:iprint>HelloWorld HelloWorld
```

You can enable Individual tracepoints like this:

```
java -Xtrace:iprint={HelloWorld(0,1)} HelloWorld
```

For details, see Table 13 on page 287.

## 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.

## Using the Trace API

There are a number of ways that you can dynamically control trace 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.

## Controlling the trace

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();
```

This method causes all active trace buffers to be written to a unique filename. 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.

---

## Chapter 34. Using the Reliability, Availability, and Serviceability Interface

The JVMRI interface will be deprecated in the near future and replaced by JVMTI extensions.

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. You can use the interface to:

- Determine the trace capability that is present
- Set and intercept trace data
- Produce various dumps
- Inject errors

You need some programming skills 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 book 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 308
- “API calls provided by JVMRI” on page 308
- “RasInfo structure” on page 314
- “RasInfo request types” on page 315
- “Intercepting trace data” on page 315
- “Calling external trace” on page 316
- “Formatting” on page 316

---

### Preparing to use JVMRI

Before you can use the JVMRI, enable the trace engine using the **-Xtrace** option along with any relevant options. See Appendix D, “Command-line options,” on page 361 for more information.

### Writing an agent

The following 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)
```

## Writing an agent

```
{  
    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 to inform the object of the tracepoints for which it should listen. See Appendix D, “Command-line options,” on page 361 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, 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 it calls.

The following example is used to register the above trace listener.

```
/*
 * Register the trace listener
 */

rc = jvmri_intf->TraceRegister(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 operation with a trace listener, including formatting the trace point information yourself then displaying this or perhaps recording it in a file or database

## 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`.

## Launching the agent

To launch 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

### 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:

```
c1 /MD /Femyagent.dll myagent.c /link /DLL
```

## Launching the agent

### Linux

To build a JVMRI agent, write a shell script similar to this:

```
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.

### z/OS

To build a JVMRI agent, write a shell script similar to this:

```
SDK_BASE= < sdk directory>
USER_DIR= < user agent's source directory>
c++ -c -g -I$SDK_BASE/include -I$USER_DIR -W "c, float(ieee)"
      -W "c, langlvl(extended)" -W "c, expo,dll" myagent.c
c++ -W "l,dll" -o libmyagent.so myagent.o
chmod 755 libmyagent.so
```

This builds a non-xplink library.

## Agent design

The agent must reference the header files jni.h and jvmri.h, which are shipped with the SDK and are in the sdk\include subdirectory. To launch 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 = jvmras_intf->TraceRegister(env, listener);
    .....
}
```

---

## API calls provided by JVMRI

The functions are not listed in the sequence in which they appear in the table. Note that all calls must be made using a valid **JNIEnv** pointer as the first parameter.

### 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**

Deregisters a dump call back function that was previously registered by a call to DumpRegister.

**Parameters**

- A valid pointer to a JNIEnv.
- Function pointer to trace function to register.

**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 trace function to register.

**Returns**

JNI return codes JNI\_OK and JNI\_ENOMEM.

**DynamicVerbosegc**

```
void JNICALL *DynamicVerbosegc (JNIEnv *env, int vgc_switch,
    int vgcon, 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 314.

**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 314.)

### 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.

### 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.

**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 33, “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 33, “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 codepage 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 codepage 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, the `TraceRegister()` routine must register a trace listener. 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. Current possible values for each character position:

0x01	One character
0x02	Short
0x04	Int
0x08	Double or long long
0xff	ASCII string pointer (may 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. It consists of a flat ASCII or EBCDIC file of the following format:

```
1.2
dg
000100 8 00 0 N DgTrcRecordsLost "***** %d records lost ****"
000101 0 00 0 N dgTraceLock_Event1 "dgTraceLock() Trace suspended and locked"
000102 0 00 0 N dgTraceUnlock_Event1 "dgTraceUnlock() Trace resumed and unlocked"
```

The first line contains the version number of the format file. A new version number reflects changes to the layout of this file. The second line contains the internal JVM component name. Following the component name are the tracepoint formatting records for the named component. These formatting records continue until another component name is found. (Component name entries can be distinguished from format records, because they always contain only one field.)

The format of each tracepoint entry is as follows:

```
nnnnnn t o l e symbolic_name .tracepoint_formatting_template
```

where:

- nnnnnn is the hex tracepoint ID.
- 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).
- symbolic\_name is the name of the tracepoint.

- `tracepoint_formatting_template` is the template in double quotes that is used to format the entry in double quotes.

Tracepoint types are as follows:

<b>Type 0</b>	Event
<b>Type 1</b>	Exception
<b>Type 2</b>	Entry
<b>Type 4</b>	Exit
<b>Type 5</b>	Exit-with-Exception
<b>Type 6</b>	Mem

Any other type is reserved for development use; you should not find any on a retail build. Note that this condition is subject to change without notice. The version number will be different for each change.

## RasInfo, trace, and formatting

## Chapter 35. 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 CPU resources. HPROF is one of the nonstandard extensions to **java**, and is invoked like this:

```
java -Xrunhprof[<option>=<value>, ...] <classname>
```

When you run Java with HPROF, an output file is created at the end of program execution. 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 filename has been given. This file contains a number of different sections, but the exact format and content depend on the selected options.

The command `java -Xrunhprof:help` displays 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 outputs information that is useful in determining where the CPU 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 "timing", you receive precise measurements of how many times each method was called and how long each execution took. Although this is more accurate, it slows down the program. If **cpu** is set to "old", the profiling data is output in the old hprof format. For more information, go to <http://java.sun.com/j2se/>, and follow the appropriate links.

**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, so getting information on 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 is very useful for detecting deadlocks.

**format=a | b**

The default is for the output file to be in ASCII format. Set **format** to 'b' if you want to specify a binary format (which is required for some utilities such as the Heap Analysis Tool).

## HPROF profiler

```
| file=<filename>
|   The file option lets you 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
|   Normally, the default (force=y) overwrites any existing information in the
|   output file. So, if you have multiple VMs running with HPROF enabled, you
|   should use force=n, which will append additional characters to the output
|   filename 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 it is not clear which thread is associated with which
|   trace (a problem that 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 represent an extremely small percentage of the code's
|   total execution time. By default, HPROF includes all execution paths that
|   represent at least 0.0001 percent of the processor's time, but 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 would specify cutoff=0.0025.
|
| verbose=y | n
|   This option causes a message to be output when dumps are taken. The default
|   is "y".
|
| lineno=y | n
|   Each frame normally includes the line number that was executed, but you can
|   use this option to suppress the line numbers from the output listing. When you
|   do so, each frame will contain 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)
```

For more information about HPROF, see <http://java.sun.com/developer/technicalArticles/Programming/HPROF.html>.

## 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 appears next.

You can see the output after using HPROF with a simple program, as shown below. This test program creates and runs two threads for a short time. From the output, you can see that the two threads called respectively "apples" and "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, filename, and line number. In the example above 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 output that looks like this:

```
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:

## HPROF profiler

- R — Runnable
- S — Suspended
- CW — Condition Wait
- MW — Monitor Wait

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 is a list of 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 to cross reference the trace output and see when this is called. If you refer back to that particular trace, you can get the line number of code that led to the creation of this object. The addresses of the constants in this class are also displayed and, in the above example, the address of the class definition for the superclass. Both classes are children of the same superclass (with address 111504e8). 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 111afb8
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" ("dump" and "sites"), you also get a list of each area of storage allocated by your code. This list is ordered with the sites that allocate the most memory at the top:

```
SITES BEGIN (ordered by live bytes) Thu Aug 29 16:30:31 2002
percent      live      alloc'ed   stack   class
rank    self   accum   bytes   objs   bytes   objs   trace   name
```

1	18.18%	18.18%	32776	2	32776	2	1332	[C]
2	9.09%	27.27%	16392	2	16392	2	1330	[B]
3	8.80%	36.08%	15864	92	15912	94	1	[C]
4	4.48%	40.55%	8068	1	8068	1	31	[S]
5	4.04%	44.59%	7288	4	7288	4	1130	[C]
6	3.12%	47.71%	5616	36	5616	36	1	<Unknown>
7	2.51%	50.22%	4524	29	4524	29	1	java/lang/Class
8	2.05%	52.27%	3692	1	3692	1	806	[L<Unknown>;
9	2.01%	54.28%	3624	90	3832	94	77	[C]
10	1.40%	55.68%	2532	1	2532	1	32	[I]
11	1.37%	57.05%	2468	3	2468	3	1323	[C]
12	1.31%	58.36%	2356	1	2356	1	1324	[C]
13	1.14%	59.50%	2052	1	2052	1	95	[B]
14	1.02%	60.52%	1840	92	1880	94	1	java/lang/String
15	1.00%	61.52%	1800	90	1880	94	77	java/lang/String
16	0.64%	62.15%	1152	10	1152	10	1390	[C]
17	0.57%	62.72%	1028	1	1028	1	30	[B]
18	0.52%	63.24%	936	6	936	6	4	<Unknown>
19	0.45%	63.70%	820	41	820	41	79	java/util/Hashtable\$Entry

In this example, Trace 1332 allocated 18.18% of the total allocated memory. This works out to be 32776 bytes.

The **cpu** option gives profiling information about the CPU. If **cpu** is set to samples, the output contains the results of periodic samples during execution of the code. At each sample, the code path being executed is recorded and a report such as this is output:

```
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 CPU execution time and was being executed 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 36. Using the JVMTI

JVMTI is a two-way interface that allows communication between the JVM and a native agent. It supersedes the deprecated JVMDI and JVMPPI interfaces, although JVMPPI-based agents such as Jprobe, OptimizeIt, TrueTime, and Quantify continue to work with the Version 5.0 JVM.

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, as well as 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, so 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 35, "Using the HPROF Profiler," on page 319

JVMTI agents can be loaded at startup using short or long forms of the command-line option:

```
-agentlib:<agent-lib-name>=<options>
```

or

```
-agentpath:<path-to-agent>=<options>
```

For example:

```
-agentlib:hprof=<options>
```

assumes that a folder containing hprof.dll is on the library path, or

```
-agentpath:C:\sdk\jre\bin\hprof.dll=<options>
```

For more information about JVMTI, see <http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html>.

For advice on porting JVMPPI 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>.



## Chapter 37. Using DTFJ

The Diagnostic Tooling Framework for Java (DTFJ) is an IBM Java application programming interface (API) used to support the building of Java diagnostics tools.

You process the dumps passed to DTFJ with the jextract tool; see “jextract” on page 225. 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.

The DTFJ API helps diagnostics tools access the following (and more) information:

- Memory locations stored in the dump
- Relationships between memory locations and Java internals
- Java threads running within the JVM
- Native threads held in the dump
- Java classes and objects that were present in the heap
- Details of the machine 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, 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.

This chapter describes DTFJ in:

- “Overview of the DTFJ interface”
- “DTFJ example application” on page 331

The full details of the DTFJ Interface are provided with the SDK as Javadoc in sdk/docs/apidoc.zip. DTFJ classes are accessible without modification to the class path.

### Overview of 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 various JVMs. All implementations support the same interface and therefore a diagnostic tool that works against a Version 1.4.2 dump will generally work with a Version 5.0 dump unless it is using knowledge of Version 1.4.2 specific internals in some way.

Figure 13 on page 330 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.

The following example shows how to work with a dump.

```
import java.io.File;
import java.util.Iterator;
import java.io.IOException;
```

## Overview of the DTFJ interface

```
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 – it would be a straightforward task to amend this code to cope with handling different implementations.

The getImage() methods in ImageFactory expect one file, the dumpfilename.zip file produced by JExtract. 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 would be 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 would be 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 Figure 13 on page 330.

## Overview of the DTFJ interface

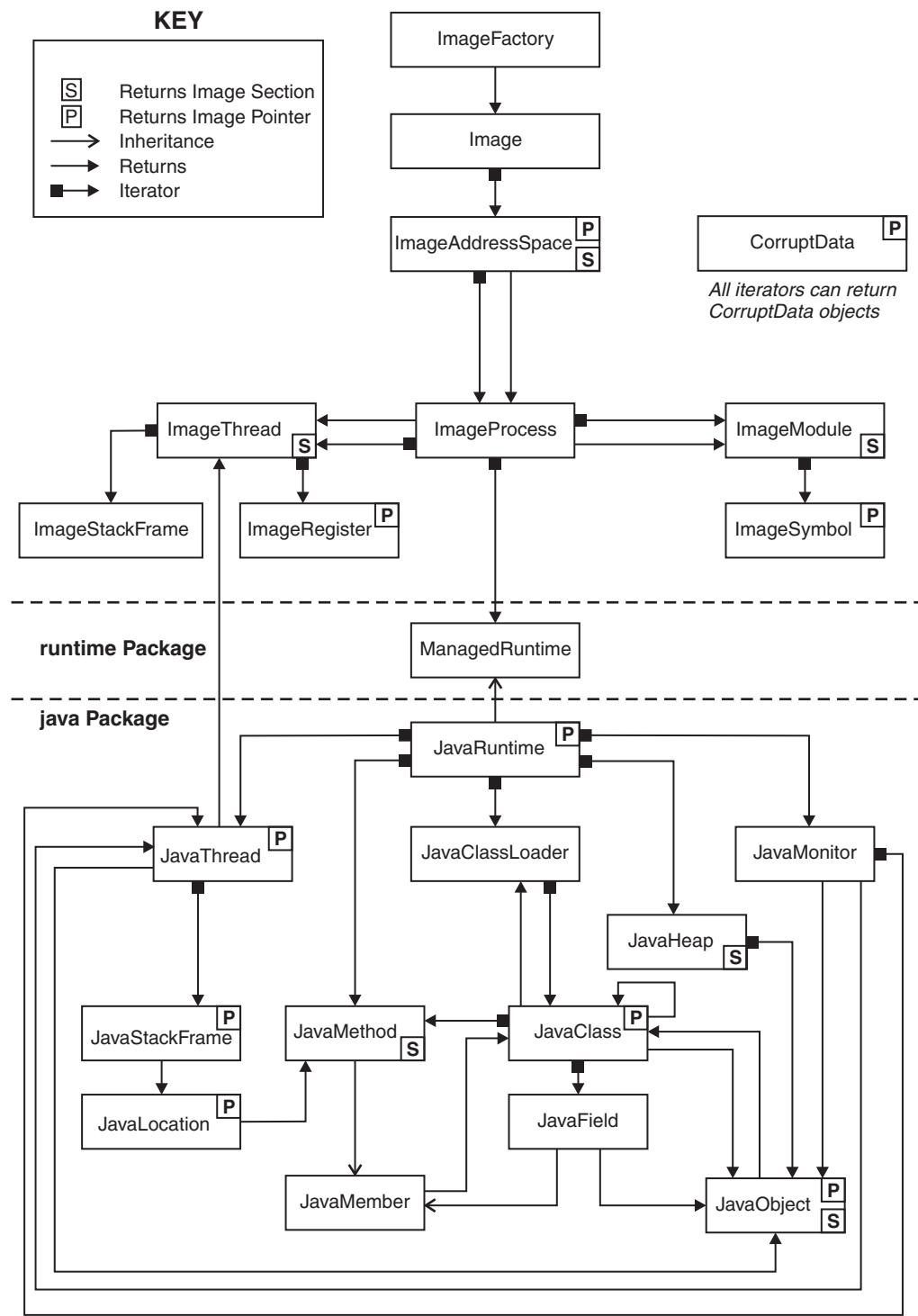


Figure 13. Diagram of the DTFJ interface

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.

## DTFJ example application

This example is a fully working DTFJ application. For clarity, it 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 would use the techniques illustrated in the example in the “Overview of the DTFJ interface” on page 327.

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.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);
            }
        }
    }
}
```

## DTFJ example application

```
| } 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.getRuntime();
            while (runTimesIt.hasNext()) {
                System.out.println("Found Runtime...");
                JavaRuntime javaRT = (JavaRuntime) runTimesIt.next();
|
                Iterator javaThreadIt = javaRT.getThreads();
|
                while (javaThreadIt.hasNext()) {
                    Object tempObj = javaThreadIt.next();
                    /* Should use CorruptData
                     * handling for all iterators
                     */
                    if (tempObj instanceof CorruptData) {
                        System.out.println("We have some corrupt data");
                    } else {
                        JavaThread javaThread = (JavaThread) tempObj;
                        System.out.println("Found JavaThread...");
                        try {
                            imgThread = (ImageThread) javaThread
                                .getImageThread();
|
                            // Now we have a Java thread we can iterator
                            // through the image threads
                            Iterator imgThreadIt = process.getThreads();
|
                            while (imgThreadIt.hasNext()) {
                                ImageThread imgThread2 = (ImageThread) imgThreadIt
                                    .next();
                                if (imgThread.equals(imgThread2)) {
                                    System.out.println("Found a match:");
                                    System.out.println("javaThread "
                                        + javaThread.getName()
                                        + " is the same as "
                                        + imgThread2.getID());
                                }
                            }
                        } catch (CorruptDataException e) {
                            System.err.println("ImageThread was corrupt: " + e.getMessage());
                        }
                    }
                }
            }
        }
    }
}
```

```
|           }
|           }
|           }
|           }
| }
```

Many DTFJ applications will follow much the same model.



---

## **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 IBM Java ORB uses. (See “Completion status and minor codes” on page 173 for more information about minor codes.)

When an error occurs, you might find additional details in the ORB FFDC log. By default, the IBM Java ORB creates an FFDC log whose name is of the form orbtrc.DDMMMYY.HHMM.SS.txt. If the IBM Java 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.

---

### 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).

**Applicable CORBA exception class:**  
org.omg.CORBA.TRANSIENT

**User response:** As with all TRANSIENT exceptions, a retry or restart of the client or server might solve the problem. Ensure that the port and server host names are correct, and that the server is running and allowing connections. Also ensure that no firewall is blocking the connection, and that a route is available between client and server.

---

### CONN\_CLOSE\_REBIND

**Explanation:** An attempt has been made to write to a TCP/IP connection that is closing.

**Applicable CORBA exception class:**  
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

**Explanation:** An unrecoverable error occurred on a TCP/IP connection. All outstanding requests are canceled. Errors include:

- A GIOP MessageError or unknown message type
- An IOException that is received while data was being read from the socket
- An unexpected error or exception that occurs during message processing

**Applicable CORBA exception class:**

---

### org.omg.CORBA.COMM\_FAILURE

**User response:** Investigate each request and reissue if necessary. If the problem reoccurs, run with ORB, network tracing, or both, active to determine the cause of the failure.

---

### CREATE\_LISTENER\_FAILED

**Explanation:** An exception occurred while a TCP/IP listener was being created.

**Applicable CORBA exception class:**  
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

**Explanation:** The server has no knowledge of the object for which the client has asked in a locate request.

**Applicable CORBA exception class:**  
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

**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.

---

**Applicable CORBA exception class:**  
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

**Explanation:** A servant failed to connect to a server-side ORB.

**Applicable CORBA exception class:**  
org.omg.CORBA.OBJ\_ADAPTER

**User response:** See the FFDC log for the cause of the problem, then try restarting the application.

---

#### 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 (whose 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>.

**Applicable CORBA exception class:**  
org.omg.CORBA.TRANSIENT

**User response:** Put the POA Manager into the active state if you want requests to be processed.

---

#### 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.

**Applicable CORBA exception class:**  
org.omg.CORBA.NO\_RESPONSE

**User response:** None.

---

#### TRANS\_NC\_LIST\_GOT\_EXC

**Explanation:** An exception was caught in the NameService while the NamingContext.List() method was executing.

**Applicable CORBA exception class:**  
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

**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.

**Applicable CORBA exception class:**  
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

**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.

**Applicable CORBA exception class:**  
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 provides the following information about environment variables:

- “Displaying the current environment”
- “Setting an environment variable”
- “Separating values in a list”
- “JVM environment settings”
- “z/OS environment variables” on page 343

---

### Displaying the current environment

To show the current environment, run:

```
set (Windows)  
env (Linux)  
set (z/OS)
```

To show a particular environment variable, run:

```
echo %ENVNAME% (Windows)  
echo $ENVNAME (Linux and z/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

To set the environment variable **LOGIN\_NAME** to *Fred*, run:

```
set LOGIN_NAME=Fred (Windows)  
export LOGIN_NAME=Fred (UNIX ksh or bash shells)
```

These variables are set only for the current shell or command-line session.

---

### Separating values in a list

If the value of an environment variable is to be a list:

- On UNIX the separator is normally a colon (:).
- On Windows the separator is usually a semicolon (;).

---

### JVM environment settings

Table 18 on page 340 summarizes common environment settings. It is not a definitive guide to all the settings. Also, the behavior of individual platforms might vary. Refer to individual sections for a more complete description of behavior and availability of these variables.

## Environment variables

*Table 18. JVM environment settings — general options*

Variable Name	Variable Values	Notes
CLASSPATH	A list of directories for the JVM to find user class files, paths, or both to individual .jar or .zip files that contain class files; for example, /mycode:/utils.jar (UNIX), D:\mycode;D:\utils.jar (Windows)	Any classpath that is set in this way is completely replaced by the -cp or -classpath Java argument if used.
IBM_JAVA_COMMAND_LINE	Set by the JVM after it starts, to enable you to find the command-line parameters set when the JVM started.	This is not available if the JVM is invoked using JNI.
IBM_JAVA_OPTIONS	This variable can be used to store default Java options. These can include -X, -D or -verbose:gc style options; for example, -Xms256m -Dibm.jvm.trusted.middleware.class.path	<p>Any options are overridden by equivalent options that are specified when Java is started.</p> <p>Does not support '-showversion'.</p> <p>Note that if you specify the name of a trace output file either directly, or indirectly, through a properties file, that output file might be accidentally overwritten if you run utilities such as the trace formatter, dump extractor, or dump formatter. For information about how to avoid this problem, see Controlling the trace, Note 2 on page 286.</p>
JAVA_ASSISTIVE	To prevent the JVM from loading Java Accessibility support, set the <b>JAVA_ASSISTIVE</b> environment variable to OFF.	
JAVA_FONTS	Define the font directory.	
JAVA_HIGH_ZIPFDS	<p>The X Windows 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 <b>JAVA_HIGH_ZIPFDS</b> environment variable to tell the JVM to use higher file descriptors for jar files. Set it to a value between 0 and 512. The JVM will then open the first jar files using file descriptors up to 1024. For example, if your program is likely to load 300 jar files:</p> <pre>export JAVA_HIGH_ZIPFDS=300</pre> <p>The first 300 jar files will then be loaded using the file descriptors 724 to 1023. Any jar files opened after that will be opened in the normal range</p>	Linux only.
JAVA_MMAP_MAXSIZE	Specifies the maximum size of zip or jar files in MB for which the JVM will use memory mapping to open those files. Files below this size are opened with memory mapping; files above this size with normal I/O.	Default=0. This default disables memory mapping.
JAVA_PLUGIN_AGENT	Specify the version of Mozilla	Linux and z/OS only.

Table 18. JVM environment settings — general options (continued)

Variable Name	Variable Values	Notes
JAVA_PLUGIN_REDIRECT	If this variable is set to a non-null value, JVM output, while serving as a plug-in, is redirected to files. The standard output and error are redirected to files plugin.out and plugin.err respectively.	Linux and z/OS only.
JAVA_ZIP_DEBUG	Setting this to any value displays memory map information as it is created.	
LANG	Specify a locale to use by default.	Linux and z/OS only.
LD_LIBRARY_PATH	This variable contains a colon-separated list of the directories from where system and user libraries are loaded. You can change which versions of libraries are loaded, by modifying this list.	Linux only.
LIBPATH	This variable contains a colon-separated list of the directories from where system and user libraries are loaded. You can change which versions of libraries are loaded, by modifying this list.	AIX and z/OS only.
PLUGIN_HOME	Set the Java plug-in path	AIX only.
SYS_LIBRARY_PATH	Specify the library path.	Linux and z/OS only.

Table 19. Deprecated JIT options

Variable Name	Variable Values	Notes
IBM_MIXED_MODE_THRESHOLD	IBM_MIXED_MODE_THRESHOLD is deprecated. In its place set IBM_JAVA_OPTIONS=-Xjit:count=<value>	See Appendix D, “Command-line options,” on page 361 for information on Java command line options. See Chapter 29, “JIT problem determination,” on page 243 for information on the -Xjit options.
JAVA_COMPILER	JAVA_COMPILER is deprecated. In its place set IBM_JAVA_OPTIONS=-Xint.	-Xint will force interpreted code, which is the same as using JAVA_COMPILER=NONE. See Appendix D, “Command-line options,” on page 361 for information on Java command line options.

Table 20. Javadump and Heapsdump options

Variable Name	Variable Values	Notes
DISABLE_JAVADUMP	Disables Javadump creation by setting to true (case-sensitive).	Use command-line option <b>-Xdisablejavadump</b> instead. Avoid using this environment variable because it makes it more difficult to diagnose failures.  On z/OS, use JAVA_DUMP_OPTS in preference.

## Environment variables

Table 20. Javadump and Heapdump options (continued)

Variable Name	Variable Values	Notes
IBM_HEAPDUMP or IBM_HEAP_DUMP	If set to 0 or FALSE, heapdump is disabled. If set to anything else, heapdump is enabled for crashes or user signals. If unset, heapdump is not enabled for crashes or user signals.	See Chapter 25, "Using Heapdump," on page 213.
IBM_HEAPDUMP_OUTOFMEMORY	When set to TRUE or 1 - generates a heapdump each time an out-of-memory exception is thrown, even if it is handled. When set to FALSE or 0 - heapdumps are not generated for out-of-memory exception. When not set - generates a heapdump when an out-of-memory exception is not caught and handled by the application.	
IBM_HEAPDUMPDIR	Specify an alternative location for Heapdump files.	On z/OS, <b>_CEE_DMPTARG</b> is used instead.
IBM_NOSIGHANDLER	Disables Java dump creation by setting to true.	Equivalent to <b>-Xrs:all</b>
IBM_JAVACOREDIR	Specify an alternative location for Javadump files; for example, on Linux IBM_JAVACOREDIR=/dumps	On z/OS <b>_CEE_DMPTARG</b> is used instead.
IBM_JAVADUMP_OUTOFMEMORY	When set to TRUE or 1 - generates a Javadump each time an out-of-memory exception is thrown, even if it is handled. When set to FALSE or 0 - Javadumps are not generated for out-of-memory exception. When not set - generates a Javadump when an out-of-memory exception is not caught and handled by the application.	
JAVA_DUMP_OPTS	Controls how diagnostic data are dumped.	The recommended way of controlling the production of diagnostic data is the <b>-Xdump</b> command-line option, described in Chapter 23, "Using dump agents," on page 195. However, if you do use JAVA_DUMP_OPTS, the recommended default value is: <b>JAVA_DUMP_OPTS="ONERROR (JAVADUMP,SYSDUMP) ONEXCEPTION (JAVADUMP,SYSDUMP), ONDUMP (JAVADUMP)"</b> For a fuller description of JAVA_DUMP_OPTS and variations across different platforms, see Chapter 26, "Using core (system) dumps," on page 217.
TMPDIR	Specify an alternative temporary directory. This is used only in the case when Javadumps and Heapdumps cannot be written to their target directory, or the current working directory	Defaults to /tmp on UNIX and C:\Temp on Windows.

Table 21. Diagnostics options

Variable Name	Variable Values	Notes
IBM_COREDIR	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.
IBM_JVM_DEBUG_PROG	Launches the JVM under the specified debugger.	Linux only.
IBM_MALLOCTRACE	Setting this variable to a non-null value enables the tracing of memory allocation in the JVM. Use with the <b>-Dcom.ibm.dbgmalloc=true</b> system property to trace native allocations from the J2SE classes.	Equivalent to <b>-memorycheck</b> .
IBM_USE_FLOATING_STACKS	Override automatic disabling of floating stacks. See the <i>Linux SDK and Runtime User Guide</i> . If not set, the launcher might set <b>LD_ASSUME_KERNEL=2.2.5</b> .	Linux only
IBM_XE_COE_NAME	This environment variable generates a system dump when the specified exception occurs. The value supplied is the package description of the exception; for example, <code>java/lang/InternalError</code>	
JAVA_PLUGIN_TRACE	To take a Java plug-in trace, set <b>JAVA_PLUGIN_TRACE=1</b> in a session in which the application will be run. This setting produces traces from both the Java and Native layer.	By default, this setting is disabled.

## z/OS environment variables

### JAVA\_DUMP\_OPTS

See Chapter 26, “Using core (system) dumps,” on page 217 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%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.

## **Environment variables**

### **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 chapter lists error messages in numeric sequence in:

- “DUMP messages”
- “J9VM messages” on page 346
- “SHRC messages” on page 349

These messages, error codes, and exit codes are generated by the JVM.

If the JVM fills all 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.

The message appendix is not complete. It will be enlarged in future editions.

### DUMP messages

#### JVMDUMP000E Dump Option unrecognised: -Xdump:%s

**Explanation:** The syntax of a -Xdump option is incorrect. The insert identifies the bad syntax.

**System action:** JVM Initialisation terminates.

**User response:** Restart the JVM after correcting or removing the -Xdump option identified.

will be caused by use of "label=" on the -Xdump.

**System action:** JVM continues

**User response:** Correct the invalid invocation.

#### JVMDUMP001E Dump Event unrecognised: ...%

**Explanation:** An invalid event name (&1) has been specified on a -Xdump events= sub-option for the JVM invocation

**System action:** JVM Initialisation terminates.

**User response:** Restart the JVM after correcting or removing the invalid events= keyword. Note: java -Xdump:events displays the valid event keywords.

#### JVMDUMP005E Missing Executable

**Explanation:** There is no executable specified on the -Xdump:tool option.

**System action:** JVM terminates.

**User response:** Check your JVM initiation to verify the syntax of any -Xdump:tool options. Note: java -Xdump:tool:events=...,exec= would cause this.

#### JVMDUMP006I Processing Dump Event "%s", detail "%.\*s" - Please Wait.

**Explanation:** A dump agent has been triggered after the JVM has encountered the specified event.

**System action:** Nothing, this is just an informational message.

**User response:** Nothing.

#### JVMDUMP007I JVM Requesting %s Dump using '%s'

**Explanation:** The JVM has initiated dump processing for a dump of type &1 for a detected event (see message JVMDUMP006). &2 describes where the dump will be generated to (normally a file) or in the case of -Xdump:tool what command line will be used. Note: There may be multiple messages generated if one event triggers several different types of dump.

**System action:** Nothing, this is just an informational message.

#### JVMDUMP002W Label Field unrecognised: %%%c

**Explanation:** The syntax of the label= option of a -Xdump startup option contains one or more invalid fields. Typically this will be caused by use of an invalid substitution character (one beginning with a %) in the label option.

**System action:** JVM continues

**User response:** On the next run of the jvm remove or correct the label= option.

#### JVMDUMP004E Missing Filename

**Explanation:** The syntax of the label= option of a -Xdump startup option contains nothing. Typically this

## DUMP messages

<b>User response:</b> Nothing.	<b>JVMDUMP014E VM Action unrecognised: ...%s</b>
<b>JVMDUMP009E %s Dump not available</b>	<b>Explanation:</b> An invalid request value (&1) has been specified on a -Xdump request= sub-option for the JVM invocation.
<b>System action:</b> JVM Continues	<b>System action:</b> The JVM terminates.
<b>User response:</b> None	<b>User response:</b> Restart the JVM after correcting or removing the invalid request= keyword. Note: java -Xdump:request displays the valid keywords.
<b>JVMDUMP010I %s Dump written to %s</b>	
<b>Explanation:</b> This is a follow on to JVMDMP007 and confirms that a dump of type &1 has been successfully written to location &2.	
<b>System action:</b> Nothing, this is just an informational message.	
<b>User response:</b> None.	
<b>JVMDUMP011I %s Dump spawned process %d</b>	
<b>Explanation:</b> The specification of a Tool dump (-Xdump:tool) has generated a new process. &2 specifies the process id of the spawned process.	
<b>System action:</b> Nothing, this is just an informational message.	
<b>User response:</b> Nothing, this is just an informational message.	
<b>JVMDUMP012E Error in %s Dump: %s</b>	
<b>Explanation:</b> The specification of a Tool dump (-Xdump:tool) attempted to spawn a new process, however the operation failed.	
<b>System action:</b> The JVM continues, however the expected tool process will not be available.	
<b>User response:</b> See the previous JVMDUMP007 message, this will display the command line that was to be used. Check the -Xdump:tool invocation (-Xdump:what may also be helpful) for correctness.	
<b>JVMDUMP013I Processed Dump Event "%s", detail "%.*s".</b>	
<b>Explanation:</b> This message accompanies JVMDUMP006 and confirms that the initiated dump processing for an event (&1) has finished.	
<b>System action:</b> Nothing, this is just an informational message.	
<b>User response:</b> None.	
<b>J9VM messages</b>	

JVMJ9VM000 Malformed value for IBM_JAVA_OPTIONS	JVMJ9VM005E Invalid value for environment variable: %s
<p><b>Explanation:</b> 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.</p> <p><b>System action:</b> The JVM terminates.</p> <p><b>User response:</b> Check the syntax of the environment variable IBM_JAVA_OPTIONS</p>	<p><b>Explanation:</b> The identified environment variable has been given an invalid value.</p> <p><b>System action:</b> The JVM terminates.</p> <p><b>User response:</b> Correct the specified environment variable and retry.</p>
JVMJ9VM001 Malformed value for -Xservice	JVMJ9VM006E Invalid command-line option: %s
<p><b>Explanation:</b> 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.</p> <p><b>System action:</b> The JVM terminates.</p> <p><b>User response:</b> Check the syntax of the -Xservice option</p>	<p><b>Explanation:</b> The identified command line option is invalid.</p> <p><b>System action:</b> The JVM terminates.</p> <p><b>User response:</b> Correct or remove the command line option and retry.</p>
JVMJ9VM002 Options file not found	JVMJ9VM007E Command-line option unrecognised: %s
<p><b>Explanation:</b> The options file specified using -Xoptionsfile couldn't be found.</p> <p><b>System action:</b> The JVM terminates.</p> <p><b>User response:</b> Correct the -Xoptionsfile option on the commandline and retry.</p>	<p><b>Explanation:</b> The identified command line option is not recognised as valid.</p> <p><b>System action:</b> The JVM terminates.</p> <p><b>User response:</b> Correct or remove the command line option and retry.</p>
JVMJ9VM003 JIT compiler "%s" not found. Will use interpreter.	JVMJ9VM008 J9VMDllMain not found
<p><b>Explanation:</b> The value specified using the -Djava.compiler option is not valid.</p> <p><b>System action:</b> The JVM continues but without a compiler (note this is generally slower than with a compiler).</p> <p><b>User response:</b> Correct the -Djava.compiler specification and retry.</p>	<p><b>Explanation:</b> J9VMDllMain is the main module entry point for system libraries (dlls). If not found then the module is unusable.</p> <p><b>System action:</b> The JVM Terminates.</p> <p><b>User response:</b> Contact your IBM Service representative.</p>
JVMJ9VM004E Cannot load library required by: %s	JVMJ9VM009 J9VMDllMain failed
<p><b>Explanation:</b> 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.</p> <p><b>System action:</b> The JVM terminates.</p> <p><b>User response:</b> 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.</p>	<p><b>Explanation:</b> J9VMDllMain is the main module entry point for system libraries (dlls). There has been a failure in its use.</p> <p><b>System action:</b> The JVM Terminates.</p> <p><b>User response:</b> Contact your IBM Service representative.</p>
JVMJ9VM010W Failed to initialize %s	
	<p><b>Explanation:</b> The identified library couldn't be initialized. This message is generally associated with JVMPi functionality.</p> <p><b>System action:</b> The JVM continues, however the expected functionality may be affected.</p> <p><b>User response:</b> Contact your IBM Service representative.</p>

## J9VM messages

---

### 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	<b>Explanation:</b> A required library couldn't be initialized. The parameter gives the name of the library.	<b>System action:</b> The JVM terminates.	JVMJ9VM038E -Xthr: unrecognized option --' '%s'	<b>Explanation:</b> The JVM has been invoked with an unrecognized -Xthr option.	<b>System action:</b> The JVM Terminates.
JVMJ9VM035 Unable to allocate OutOfMemoryError	<b>Explanation:</b> The JVM tried to issue an OutOfMemoryError but failed. This is often indicative of a failure in the user application design/useage.	<b>System action:</b> The JVM Terminates.	JVMJ9VM039 -Xscmx is ignored if -Xshareclasses is not specified	<b>Explanation:</b> The -Xscmx'n' option is not meaningful unless shared class support is active.	<b>System action:</b> The JVM continues.
	<b>User response:</b> 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.			<b>User response:</b> Remove the -Xscmx option or activate shared classes by using -Xshareclasses.	
<b>SHRC messages</b>					
JVMSHRC004E Cannot destroy cache "%1\$s"				destroyed. Investigate these messages.	
<b>Explanation:</b> It has not been possible to destroy the named shared classes cache.			JVMSHRC008I Shared Classes Cache created: %1\$s size: %2\$d bytes	<b>Explanation:</b> This is an information message notifying you that a shared classes cache of the given name and size has been created.	
<b>System action:</b> Processing continues.			<b>System action:</b> The JVM continues.		
<b>User response:</b> Other messages may have been issued indicating the reason why the cache has not been destroyed. Investigate these messages.			<b>User response:</b> None required. This is an information message issued when verbose shared classes messages have been requested.		
JVMSHRC005 No shared class caches available			JVMSHRC009I Shared Classes Cache opened: %1\$s size: %2\$d bytes	<b>Explanation:</b> This is an information message notifying you that an existing shared classes cache of the given name and size has been opened.	
<b>Explanation:</b> There are no shared class caches present on the system which can be processed by the command requested			<b>System action:</b> The JVM continues.		
<b>System action:</b> Processing continues.			<b>User response:</b> None required. This is an information message issued when verbose shared classes messages have been requested.		
<b>User response:</b> None required.			JVMSHRC010I Shared Cache "%1\$s" is destroyed	<b>Explanation:</b> This is an information message notifying you that the named shared classes cache has been destroyed as requested.	
JVMSHRC006I Number of caches expired within last %1\$d minutes is %2\$d			<b>System action:</b> 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.		
<b>Explanation:</b> This is an information message issued by the system.			<b>User response:</b> None required. This is an information		
<b>System action:</b> Processing continues.					
<b>User response:</b> None required.					
JVMSHRC007I Failed to remove shared class cache "%1\$s"					
<b>Explanation:</b> It has not been possible to remove the indicated shared class cache.					
<b>System action:</b> Processing continues.					
<b>User response:</b> Other messages may have been issued indicating the reason why the cache has not been					

## SHRC messages

| 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 occured 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".

## SHRC messages

<b>JVMSHRC029E Not enough memory left on the system</b>	<b>JVMSHRC057 Wrong parameters for expire option</b>
<p><b>Explanation:</b> 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.</p> <p><b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.</p> <p><b>User response:</b> 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.</p>	<p><b>Explanation:</b> The value specified for the expire parameter of -Xshareclasses is invalid.</p> <p><b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.</p> <p><b>User response:</b> Rerun the command with a valid expire value. This must be a positive integer.</p>
<b>JVMSHRC030E The Shared Class Cache you are attaching has invalid header.</b>	<b>JVMSHRC058 ERROR: Cannot allocate memory for ClasspathItem in shrinit::hookStoreSharedClass</b>
<p><b>Explanation:</b> The shared classes cache you are trying to use is invalid.</p> <p><b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.</p> <p><b>User response:</b> Contact your IBM service representative.</p>	<p><b>Explanation:</b> An error has occurred in shared class processing.</p> <p><b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.</p> <p><b>User response:</b> 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.</p>
<b>JVMSHRC031E The Shared Class Cache you are attaching has incompatible JVM version.</b>	<b>JVMSHRC059 ERROR: Cannot allocate memory for ClasspathItem in shrinit::hookFindSharedClass</b>
<p><b>Explanation:</b> The shared classes cache you are trying to use is incompatible with this JVM.</p> <p><b>System action:</b> The JVM terminates.</p> <p><b>User response:</b> 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</p>	<p><b>Explanation:</b> An error has occurred in shared class processing.</p> <p><b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.</p> <p><b>User response:</b> 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.</p>
<b>JVMSHRC032E The Shared Class Cache you are attaching has wrong modification level.</b>	<b>JVMSHRC060 ERROR: Cannot allocate memory for string buffer in shrinit::hookFindSharedClass</b>
<p><b>Explanation:</b> The shared classes cache you are trying to use is incompatible with this JVM.</p> <p><b>System action:</b> The JVM terminates.</p> <p><b>User response:</b> 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</p>	<p><b>Explanation:</b> An error has occurred in shared class processing.</p> <p><b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.</p> <p><b>User response:</b> 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.</p>

**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.

## SHRC messages

---

### 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.	"-Xshareclasses:nonfatal", in which case the JVM
<b>User response:</b> 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.	continues without using Shared Classes.
<b>JVMSHRC083 ERROR: Cannot allocate memory for identifiedClasspaths array in SH_ClasspathManagerImpl2</b>	<b>User response:</b> 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.
<b>Explanation:</b> An error has occurred in shared class   processing.	<b>JVMSHRC087 ERROR: MarkStale failed during ClasspathManager::update()</b>
<b>System action:</b> The JVM terminates, unless you have   specified the nonfatal option with   "-Xshareclasses:nonfatal", in which case the JVM   continues without using Shared Classes.	<b>Explanation:</b> An error has occurred in shared class   processing.
<b>User response:</b> 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.	<b>System action:</b> The JVM terminates, unless you have   specified the nonfatal option with   "-Xshareclasses:nonfatal", in which case the JVM   continues without using Shared Classes.
<b>JVMSHRC084 ERROR: Cannot allocate memory for linked list item</b>	<b>User response:</b> Contact your IBM service   representative.
<b>Explanation:</b> An error has occurred in shared class   processing.	<b>JVMSHRC088 ERROR: Failed to create cache as ROMImageSegment in SH_CacheMap</b>
<b>System action:</b> The JVM terminates, unless you have   specified the nonfatal option with   "-Xshareclasses:nonfatal", in which case the JVM   continues without using Shared Classes.	<b>Explanation:</b> An error has occurred in shared class   processing.
<b>User response:</b> 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.	<b>System action:</b> The JVM terminates, unless you have   specified the nonfatal option with   "-Xshareclasses:nonfatal", in which case the JVM   continues without using Shared Classes.
<b>JVMSHRC085 ERROR: Cannot allocate memory for linked list item header</b>	<b>User response:</b> Contact your IBM service   representative.
<b>Explanation:</b> An error has occurred in shared class   processing.	<b>JVMSHRC089 ERROR: Cannot create refresh mutex in SH_CacheMap</b>
<b>System action:</b> The JVM terminates, unless you have   specified the nonfatal option with   "-Xshareclasses:nonfatal", in which case the JVM   continues without using Shared Classes.	<b>Explanation:</b> An error has occurred in shared class   processing.
<b>User response:</b> 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.	<b>System action:</b> The JVM terminates, unless you have   specified the nonfatal option with   "-Xshareclasses:nonfatal", in which case the JVM   continues without using Shared Classes.
<b>JVMSHRC086 ERROR: Cannot enter ClasspathManager hashtable mutex</b>	<b>User response:</b> 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.
<b>Explanation:</b> An error has occurred in shared class   processing.	<b>JVMSHRC090 ERROR: Failed to get cache mutex in SH_CacheMap startup</b>
<b>System action:</b> The JVM terminates, unless you have   specified the nonfatal option with	<b>Explanation:</b> An error has occurred in shared class   processing.
	<b>System action:</b> The JVM terminates, unless you have   specified the nonfatal option with   "-Xshareclasses:nonfatal", in which case the JVM   continues without using Shared Classes.
<b>User response:</b> The system may be low on resources,   please retry when the system is more lightly loaded. If	<b>User response:</b> The system may be low on resources,   please retry when the system is more lightly loaded. If

## SHRC messages

the situation persists, contact your IBM service representative.	continues without using Shared Classes.
<b>JVMSHRC091 ERROR: Read corrupt data for item 0x%p (invalid dataType)</b>  <b>Explanation:</b> An error has occurred in shared class processing.  <b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.  <b>User response:</b> Contact your IBM service representative.	<b>User response:</b> Contact your IBM service representative.
<b>JVMSHRC092 ERROR: ADD failure when reading cache</b>  <b>Explanation:</b> An error has occurred in shared class processing.  <b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.  <b>User response:</b> Contact your IBM service representative.	<b>JVMSHRC096 WARNING: Shared Cache "%s" is full. Use -Xscmx to set cache size.</b>  <b>Explanation:</b> The named shared classes cache is full and no further classes may be added to it.  <b>System action:</b> 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.  <b>User response:</b> 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.
<b>JVMSHRC093 INFO: Detected unexpected termination of another JVM during update</b>  <b>Explanation:</b> The JVM has detected an unexpected termination of another JVM while updating the shared classes cache.  <b>System action:</b> The JVM continues.  <b>User response:</b> No action required, this message is for information only.	<b>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.</b>  <b>Explanation:</b> The shared classes cache named in the message is corrupt.  <b>System action:</b> The JVM continues.  <b>User response:</b> Destroy the shared classes cache named in the message and rerun. If the situation persists, contact your IBM service representative.
<b>JVMSHRC094 ERROR: Orphan found but local ROMClass passed to addROMClassToCache</b>  <b>Explanation:</b> An error has occurred in shared class processing.  <b>System action:</b> The JVM continues if possible.  <b>User response:</b> Contact your IBM service representative.	<b>JVMSHRC125 ERROR: Could not allocate memory for string buffer in SH_CacheMap</b>  <b>Explanation:</b> An error has occurred in shared class processing.  <b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.  <b>User response:</b> 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.
<b>JVMSHRC095 ERROR: Attempts to call markStale on shared cache items have failed</b>  <b>Explanation:</b> An error has occurred in shared class processing.  <b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM	<b>JVMSHRC126 ERROR: Request made to add too many items to ClasspathItem</b>  <b>Explanation:</b> An error has occurred in shared class processing.  <b>System action:</b> The JVM continues.  <b>User response:</b> Contact your IBM service representative.

JVMSHRC127 SH_CompositeCache::enterMutex failed with return code %d	JVMSHRC132 ERROR: Cannot allocate memory for hashtable entry in ROMClassManagerImpl
<p><b>Explanation:</b> An error has occurred while trying to update the shared classes cache.</p> <p><b>System action:</b> The JVM will terminate.</p>	<p><b>Explanation:</b> An error has occurred in shared class processing.</p> <p><b>System action:</b> The JVM continues.</p>
<p><b>User response:</b> 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.</p>	<p><b>User response:</b> 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.</p>
JVMSHRC128 SH_CompositeCache::exitMutex failed with return code %d	JVMSHRC133 ERROR: Cannot enter ROMClassManager hashtable mutex
<p><b>Explanation:</b> An error has occurred while trying to update the shared classes cache.</p> <p><b>System action:</b> The JVM will terminate.</p>	<p><b>Explanation:</b> An error has occurred in shared class processing.</p> <p><b>System action:</b> The JVM continues.</p>
<p><b>User response:</b> Contact your IBM service representative.</p>	<p><b>User response:</b> 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.</p>
JVMSHRC129 ERROR: Attempt to set readerCount to -1!	JVMSHRC134 ERROR: Failed to create pool in SH_ROMClassManagerImpl
<p><b>Explanation:</b> An error has occurred while trying to update the shared classes cache.</p> <p><b>System action:</b> The JVM continues.</p>	<p><b>Explanation:</b> An error has occurred in shared class processing.</p> <p><b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.</p>
<p><b>User response:</b> Contact your IBM service representative.</p>	<p><b>User response:</b> 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.</p>
JVMSHRC130 ERROR: Attempt to allocate while commit is still pending	JVMSHRC135 ERROR: Failed to create hashtable in SH_ROMClassManagerImpl
<p><b>Explanation:</b> An error has occurred while updating the shared classes cache.</p> <p><b>System action:</b> The processing will continue, if possible.</p>	<p><b>Explanation:</b> An error has occurred in shared class processing.</p> <p><b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.</p>
<p><b>User response:</b> Contact your IBM service representative.</p>	<p><b>User response:</b> 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.</p>
JVMSHRC131 ERROR: Cannot allocate memory for linked list item in ROMClassManagerImpl	
<p><b>Explanation:</b> An error has occurred in shared class processing.</p> <p><b>System action:</b> The JVM continues.</p>	
<p><b>User response:</b> 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.</p>	

## SHRC messages

<p><b>JVMSHRC136</b> ERROR: Cannot create monitor in SH_ROMClassManagerImpl</p> <p><b>Explanation:</b> An error has occurred in shared class processing.</p> <p><b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.</p> <p><b>User response:</b> 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.</p>	<p><b>JVMSHRC155</b> Error copying username into cache name</p> <p><b>Explanation:</b> The system has not been able to obtain the username for inclusion in the shared classes cache name.</p> <p><b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.</p> <p><b>User response:</b> Contact your IBM service representative.</p>
<p><b>JVMSHRC137</b> SAFE MODE: Warning: ROMClass %.*s does not match ROMClass in cache</p> <p><b>Explanation:</b> 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.</p> <p><b>System action:</b> The JVM continues.</p> <p><b>User response:</b> 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.</p>	<p><b>JVMSHRC156</b> Error copying groupname into cache name</p> <p><b>Explanation:</b> The system has not been able to obtain the groupname for inclusion in the shared classes cache name.</p> <p><b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.</p> <p><b>User response:</b> Contact your IBM service representative.</p>
<p><b>JVMSHRC147</b> Character %.*s not valid for cache name</p> <p><b>Explanation:</b> The shared classes cache name specified to the JVM contains the indicated invalid character.</p> <p><b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.</p> <p><b>User response:</b> Change the requested shared classes cache name so that all characters are valid and rerun.</p>	<p><b>JVMSHRC157</b> 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)</p> <p><b>Explanation:</b> 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.</p> <p><b>System action:</b> The JVM continues.</p> <p><b>User response:</b> If you require a larger cache: destroy this cache, increase the value of SHMMAX and recreate the cache.</p>
<p><b>JVMSHRC154</b> Escape character %.*s not valid for cache name</p> <p><b>Explanation:</b> An invalid escape character has been specified in a shared classes cache name</p> <p><b>System action:</b> The JVM terminates, unless you have specified the nonfatal option with "-Xshareclasses:nonfatal", in which case the JVM continues without using Shared Classes.</p> <p><b>User response:</b> Specify a valid escape character. Valid escape characters are %u for username (all platforms) and %g for group name (not valid for Windows).</p>	<p><b>JVMSHRC158</b> Successfully created shared class cache "%1\$s"</p> <p><b>Explanation:</b> 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.</p> <p><b>System action:</b> The JVM continues.</p> <p><b>User response:</b> No action required, this is an information only message.</p>

**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 message 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\$... ...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 message 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 appropriately.

**JVMSHRC172 AIX cannot create cache of requested size: Please refer to the Java Diagnostics Guide**

**Explanation:** AIX cannot create a Shared Classes cache of 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:** Please refer to the Java User Guide or Diagnostics Guide.

## **SHRC messages**

---

## 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”
- “System property command-line options” on page 363
- “Nonstandard command-line options” on page 364
- “JIT command-line options” on page 369
- “Garbage Collector command-line options” on page 370

---

### 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. The precedence rules (in descending order) for specifying options are:

1. Command-line.

For example, `java -X<option> MyClass`

2. A file containing a list of options, specified using the **-Xoptionsfile** option on the command line. For example, `java -Xoptionsfile=myoptionfile.txt MyClass`

In the options file, specify each option on a new line; you can use the ‘\’ character as a continuation character if you want a single option to span multiple lines. Use the '#' character to define comment lines. You cannot specify **-classpath** in an options file. Here is an example of an options file:

```
#My options file
-X<option1>
-X<option2>=\
<value1>,\
<value2>
-D<sysprop1>=<value1>
```

3. **IBM\_JAVA\_OPTIONS** environment variable. You can set command-line options using this environment variable. The options 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

#### **-assert**

Prints help on assert-related options.

#### **-cp, -classpath<directories and zip or jar files separated by ;> (or : on Linux and z/OS)**

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 (.).

## General command-line options

### **-help, -?**

Prints a usage message.

### **-showversion**

Prints product version and continues.

### **-memorycheck[:<option>]**

Identifies memory leaks inside the JVM. Options are:

- **all**

The default if just **-memorycheck** is used. Enables checking of all allocated and freed blocks on every free and allocate call. This check of the heap is the most thorough and should cause a crash on nearly all memory-related problems very soon after they are caused. This option has the greatest impact on performance.

- **quick**

Enables block padding only. Used to detect basic heap corruption. The detection is based on padding every allocated block with sentinel bytes, which are verified on every allocate and free. If any of this padding has been damaged in the allocated block or previously allocated block, the program terminates, printing a message that memory errors were found. Block padding is much less work than the usual operation, in which every block is checked. This difference results in a much faster, although not quite as effective, operation.

- **nofree**

Blocks are never freed, but are added to a list of blocks already used. These blocks are checked, along with currently allocated blocks, for memory corruption on every allocation and deallocation. This checking helps to detect a dangling pointer (a pointer that is "dereferenced" after its target memory is freed). The data portion of freed blocks is filled, similar to the padding, with 64-bit values of the form 0xBEEF0xxxxxxxxxxDDDE. This option cannot be reliably used with long-running applications (such as WAS), because "freed" memory is never reused or released by the VM.

- **failat**

Takes an additional number as a parameter (for example:

**-memorycheck:failat=13**). This option causes memory allocation to fail (return NULL) after the given number of allocations. In the given example of 13, for example, the 14th allocation will return NULL. (Note that deallocations are not factored into this comparison.) This option is mainly used to ensure that VM code reliably handles allocation failures. This option is useful for checking allocation site behavior rather than setting a specific allocation limit.

- **skipto**

Takes an additional number as a parameter (for example:

**-memorycheck:skipto=13**). This option causes the program to check only on allocations that occur after the given number of allocations have already been done. (Note that deallocations are not factored into this comparison.) This option can be useful in speeding up the operation by skipping over many of the initial VM allocations that may be assumed unlikely causes of a new problem. As a rough estimate, the VM performs 250+ allocations during startup.

- **callsite**

Takes an additional number as a parameter (for example:

**-memorycheck:callsite=1000**). This option causes the program to print the callsite information every n number of allocations. (Note that

deallocations are not factored into this number.) The callsite information printed is similar to the statistics printed on VM termination; however, the statistics are presented in a tabular format with separate statistics for each callsite. Additional 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.

Not all callsites currently provide callsite information, so statistics for all such callsites will be accumulated in an entry labelled "unknown".

- **zero**

Newly allocated blocks are memset to 0 instead of being filled with the 0xE7E7xxxxxxxxxE7E7 pattern. Setting to 0 helps you to determine whether a callsite is expecting zeroed memory (in which case the allocation request should be followed by `memset(pointer, 0, size)`).

**-verbose[:class | gc | jni]**

Enables verbose output.

**-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
- 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>
```

**-version**

Prints product version.

## System property command-line options

**-D<name>=<value>**

Sets a system property.

**-Dcom.ibm.lang.management.verbose**

Setting this property enables verbose information from `java.lang.management` operations to be written to the console during VM operation.

**-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 that which 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 }**

Disable 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

## System property command-line options

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.net.client.defaultConnectTimeout=<value in milliseconds>**

This property 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 there is no timeout set.

When a connection is made by an applet to a server and the server does not respond properly, the applet might appear to hang and might also cause the browser to hang. This 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>**

This property 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 there is no timeout set.

### **-Dsun.rmi.transport.tcp.connectionPool={true | any non-null value}**

Enables thread pooling for the RMI ConnectionHandlers in the TCP transport layer implementation.

### **-Dswing.useSystemFontSettings={false}**

From v1.4.1 onwards, by default, Swing programs running with the Windows Look and Feel render with the system font set by the user instead of a Java-defined font. As a result, fonts for v1.4.1 differ from those in prior 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 prior releases will be the same for Swing programs running with the Windows Look and Feel.

---

## Nonstandard command-line options

The **-X** options are nonstandard and subject to change without notice.

Options that relate to the JIT are listed under “JIT command-line options” on page 369. Options that relate to the Garbage Collector are listed under “Garbage Collector command-line options” on page 370.

### **-X**

Prints help on nonstandard options.

### **-Xargencoding**

Allows you to put Unicode escape sequences in the argument list. This option is set to off by default.

### **-Xbootclasspath:<directories and zip or jar files separated by ;> (or : on Linux and z/OS)**

Sets the search path for bootstrap classes and resources. The default is to search for bootstrap classes and resources within the internal VM directories and .jar files.

**-Xbootclasspath/a:<directories and zip or jar files separated by ;> (or : on Linux and z/OS)**

Appends the specified directories, zip, or jar files to the end of bootstrap class path. The default is to search for bootstrap classes and resources within the internal VM directories and .jar files.

**-Xbootclasspath/p:<directories and zip or jar files separated by ;> (or : on Linux and z/OS)**

Prepends the specified directories, zip, or jar files to the front of the bootstrap class path. Do not deploy applications that use the **-Xbootclasspath:** or

**-Xbootclasspath/p:** option to override a class in the standard API, because such a deployment would contravene the Java 2 Runtime Environment binary code license. The default is to search for bootstrap classes and resources within the internal VM directories and .jar files.

**-Xcheck:jni[:help][[:option]=<value>]**

Performs additional checks for JNI functions. (Equivalent to **-Xrunjnichk**.) By default, no checking is performed.

**-Xcheck:nabounds**

Performs additional checks for JNI array operations. You can also use

**-Xrunjnichk**. By default, no checking is performed.

**-Xdbg:<options>**

Loads debugging libraries to support remote debug applications. (Equivalent to **-Xrunjdwp**.) By default, the debugging libraries are not loaded, and the VM instance is not enabled for debug.

**-Xdbginfo:<symbol file path>**

Loads the debug info server with the symbol path specified. By default, the debug information server is disabled.

**-Xdebug**

Enables remote debugging. By default, the debugger is disabled.

**-Xdisablejavadump**

Turns off javadump generation on errors and signals. By default, javadump generation is enabled.

**-Xdump[:help] | [:<option>=<value>]**

See Chapter 23, “Using dump agents,” on page 195.

**-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 these are available. Only Java code and system native methods can be on IFA processors.

**-Xiss<size>**

Sets the initial stack size for Java threads. By default, 2 kilobytes.

**-Xlinenumbers**

Displays line numbers in stack traces, for debugging. See also **-Xnolinenumbers**. By default, line numbers are on.

## System property command-line options

**-Xlp <size>** (AIX , Windows Server 2003 (x86, AMD64, EM64T), 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 AIX standard 4 KB pages. AIX requires special configuration to enable large pages. For more information on configuring AIX support for large pages, see [http://www.ibm.com/servers/aix/whitepapers/large\\_page.html](http://www.ibm.com/servers/aix/whitepapers/large_page.html). The SDK supports the use of large pages only to back the Java heap's 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 will 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. To use large pages, the user that will run 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**, then find **Local Policies->User Rights Assignment->"Lock pages in memory"**. Add the user that will run the Java process and reboot your machine. For more information, see:

[http://www.microsoft.com/technet/prodtechnol/windowsserver2003/...](http://www.microsoft.com/technet/prodtechnol/windowsserver2003/)

[http://msdn.microsoft.com/library/en-us/memory/base/large\\_page\\_...](http://msdn.microsoft.com/library/en-us/memory/base/large_page_...)

<http://msdn.microsoft.com/library/en-us/memory/base/getlargepage...>

**-Xmso<size>**

Sets the initial stack size for operating system threads (format = nn[K|M|G]). By default, this option is set to 256 KB, except for Windows 32-bit, which is set to 32 KB.

**-Xnoagent**

Disables support for the oldjdb debugger.

**-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 defines. By default, no option file is used.

**-Xoss<size>**

Recognized but DEPRECATED. Use **-Xss** and **-Xmso**. Sets maximum Java stack size for any thread (format = nn[K|M|G]). The AIX default is 400 KB.

### **-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**

Reduces the use of operating system signals. This prevents the JVM from installing signal handlers for all but exception type signals (such as SIGSEGV, SIGILL, SIGFPE). By default, the VM makes full use of operating system signals.

**Note:** Linux always uses SIGU3R1 and SIGU3R2.

### **-Xrunhprof[:help] | [:<option>=<value>, ...]**

Performs heap, CPU, or monitor profiling.

### **-Xrunjdwp[:help] | [:<option>=<value>, ...]**

Loads debugging libraries to support remote debug applications. This is the same as **-Xdbg**.

### **-Xrunjnicchk[:help] | [:<option>=<value>, ...]**

Performs additional checks for JNI functions to trace errors in native programs that access the JVM using JNI. (Equivalent to **-Xcheck:jni**.)

### **-Xscmx<size>[k|m|g]**

Specifies cache size. This option applies only if a cache is being created and no cache of the same name exists. 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 you can specify is limited by the amount of physical memory and paging space available to the system. Because 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 will reduce the size of the shared classes cache you can create.

### **-Xshareclasses:<suboptions>**

Enables class sharing. Can take a number of suboptions, some of which are cache utilities. Cache utilities perform the required operation on the specified cache, without starting the VM. You can combine multiple suboptions, separated by commas, but the cache utilities are mutually exclusive.

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

- **help**

Lists all the command-line options.

- **name=<name>**

Connects to a cache of a given name, creating the cache if it does not already exist. Use the **listAllCaches** utility option to show which caches are currently available. If you do not specify a name, the default name "sharedcc\_" is combined with the current user name appended to create the cache name.

- **groupAccess**

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

- **verbose**

Gives detailed output on the cache I/O activity, listing information on classes being stored and found. Each classloader is given a unique ID (the bootstrap loader is always 0) and the output shows the classloader hierarchy at work, where classloaders must ask their parents for a class before they can load it themselves. It is normal to see many failed

## System property command-line options

requests; this behavior is expected for the classloader hierarchy. Note that the standard option **-verbose:class** also enables class sharing verbose output if class sharing is enabled.

- **modified=<modified context>**  
Required if a bytecode agent is installed that might modify bytecode at runtime. If you do not specify this option and a bytecode agent is installed, no classes are stored or found in the cache. The *<modified context>* is a descriptor chosen by the user; for example "myModification1". This option partitions the cache, so that only JVMs using context myModification1 can share the same classes. For instance, if you run "HelloWorld" with a modification context and then run it again with a different modification context, you will see all classes stored twice in the cache.
- **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 VMs 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 VMs 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.
- **listAllCaches** (Utility option)  
Lists all the caches on the system, describing if they are in use and when they were last used.
- **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. Note that "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 268 for more information.
- **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 along with a reference to the location from which it was loaded. See "printAllStats utility" on page 269 for more information.

### **-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 (format = nn[K|M|G]). The default is 256 KB for 32-bit platforms and 512 KB for 64-bit platforms.

### **-Xthr:<options>**

Sets the threading options.

### **-Xtrace[:help] | [:<option>=<value>, ...]**

See page “Specifying trace options” on page 286.

### **-Xverify[:<option>]**

With no parameters, enables the verifier. Note that this is the default in all J2SE JVMs; used on its own, this 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

---

## JIT command-line options

You might need to read Chapter 29, “JIT problem determination,” on page 243 to understand some of the references that are given here. The following list contains all the JIT command-line options that are available in this release.

### **-Xcodecache<size>**

This option is for performance. It sets the size of each block of memory that is allocated to store native code of compiled Java methods. By default, this size is selected internally according to the CPU architecture and the capability of your system. If profiling tools show significant costs in trampolines (JVMTI identifies trampolines in a methodLoad2 event), that is a good prompt 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 inter-block calls are eliminated. A reasonable starting point to tune for the optimal size is (totalNumberByteOfCompiledMethods \* 1.1).

### **-Xcomp (z/OS only)**

Forces methods to be compiled by the JIT 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) and Ahead-Of-Time (AOT) compilers. By default, the JIT and AOT compilers are enabled.

### **-Xjit [:<option>=<value>, ...]**

With no parameters, it enables the JIT. Note that the JIT is enabled by default in all J2SE JVMs, so using this option on its own has no effect. Use this option to control the behavior of the JIT. Useful parameters are:

- **count=<n>** – where <n> is the number of times a method is invoked before it is compiled. For example, setting count=0 forces the JIT to compile everything on first execution.
- **optlevel=[noOpt | cold | warm | hot | veryHot | scorching]** – forces the JIT to compile all methods at a specific optimization level.
- **verbose** – displays information about the JIT configuration and method compilation.

### **-Xnoaot**

Turns off AOT support (JIT still loads). By default, the AOT compiler is enabled.

### **-Xnojit**

Turns off the JIT (AOT support still loads). By default, the JIT compiler is enabled.

### **-Xquickstart**

Used for improving startup time of some Java applications. **-Xquickstart** causes

## JIT command-line options

the JIT to run with a subset of optimizations; that is, a quick compile. This quick compile allows for improved startup time. **-Xquickstart** is appropriate for shorter running applications, especially those where execution time is not concentrated into a small number of methods. **-Xquickstart** can degrade performance if it is used on longer-running applications that contain hot methods. The implementation of **-Xquickstart** is subject to change in future releases.

---

## Garbage Collector command-line options

You might need to read Chapter 2, “Understanding the Garbage Collector,” on page 7 to understand some of the references that are given here. The following lists contain all the Garbage Collector command-line options that are available in this release.

In the following list, **<value>** is an integer value.

**-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**

Compacts on all system garbage collections. Compaction takes place on global garbage collections if you specify **-Xcompactgc** or if compaction triggers are met. By default, compaction occurs only when triggered internally.

**-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. No forced compactions

**-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 which area’s usage, LOA or SOA, 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**

Signals to the VM that calls to `System.gc()` should have no effect. 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, this 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, nocompact, to the Garbage Collector.

### **-Xgcpolicy:<optthrput | optavgpause | gencon | subpool (AIX, Linux PPC and zSeries, and z/OS only) >**

Controls the behavior of the Garbage Collector.

The *optthrput* option is the default and delivers very 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 very 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 PPC and zSeries, and z/OS only) 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 of threads>**

Sets the number of helper threads that are used for parallel operations during garbage collection. By default, the number of threads is set to the number of physical CPUs present -1, with a minimum of 1.

### **-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 will be 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.

### **-Xloainitlal<number>**

<number> is a floating point number between 0 and 0.95, which specifies the initial percentage of the current tenure space allocated to the large object area (LOA). The default is 0.05 or 5%.

### **-Xloamaximum<number>**

<number> is a floating point number between 0 and 0.95, which specifies the maximum percentage of the current tenure space allocated to the large object area(LOA). The default is 0.5 or 50%.

### **-Xmaxe<value>**

Sets the maximum amount by which the garbage collector expands the heap.

## Garbage Collector command-line options

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 10MB. By default, there is no maximum expansion size.

### **-Xmaxf<value>**

Specifies the maximum percentage of heap that must be free after a garbage collection. If the free space exceeds this amount, the JVM attempts to shrink the heap. Specify the size as a decimal value in the range 0-1, for example **-Xmaxf0.5** sets the maximum free space to 50%. A value of **-Xmaxf1.0** disables heap contraction. The default value is 0.6 (60%).

### **-Xmca<value>**

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 32K.

### **-Xmco<value>**

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 128K.

### **-Xmdx<value>**

Sets the default maximum size of the memory space (**Xmdx <= Xmx**). If you do not specify **-Xresman**, **-Xmdx** is ignored.

### **-Xmine<value>**

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 by 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 50MB. By default, the minimum expansion size is 1MB.

### **-Xminf<value>**

Specifies the minimum percentage of heap that should be free after a garbage collection. If the free space falls below this amount, the JVM attempts to expand the heap. Specify the size as a decimal value in the range 0-1; for example, a value of **-Xminf0.3** requests the minimum free space to be 30% of the heap. By default, the minimum value is 0.3.

### **-Xmn<value>**

Sets the initial and maximum size of the new (nursery) heap to the specified value. Equivalent to setting both **-Xmns** and **-Xmnx**. If you set either **-Xmns** or **-Xmnx**, you cannot set **-Xmn**. If you attempt to set **-Xmn** with either **-Xmns** or **-Xmnx**, the VM will not start, returning an error. By default, **-Xmn** is selected internally according to your system's capability. 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.

### **-Xmns<value>**

Sets the initial size of the new (nursery) heap to the specified value. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmn**. If the scavenger is disabled, this option is ignored.

### **-Xmnx<value>**

Sets the maximum size of the new (nursery) heap to the specified value. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmn**. If the scavenger is disabled, this option is ignored.

### **-Xmo<value>**

Sets the initial and maximum size of the old (tenured) heap to the specified value. Equivalent to setting both **-Xmos** and **-Xmox**. If you set either **-Xmos** or **-Xmox**, you cannot set **-Xmo**. If you attempt to set **-Xmo** with either **-Xmos** or **-Xmox**, the VM will not start, returning an error. By default, **-Xmo** is selected internally according to your system's capability. You can use the **-verbose:sizes** option to find out the values that the VM is currently using.

### **-Xmoi<value>**

Sets the amount the Java heap is incremented. If set to zero, no expansion is allowed. By default, the increment size is calculated on the expansion size, **-Xmine** and **-Xminf**.

### **-Xmos<value>**

Sets the initial size of the old (tenure) heap to the specified value. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmo**.

### **-Xmox<value>**

Sets the maximum size of the old (tenure) heap to the specified value. By default, this option is selected internally according to your system's capability. This option will return an error if you try to use it with **-Xmo**.

### **-Xmr<value>**

Sets the size of the Garbage Collection "remembered set". This is a list of objects in the old (tenured) heap that have references to objects in the new (nursery) heap. By default, this option is set to 16 kilobytes.

### **-Xmrx<value>**

Sets the remembered maximum size setting

### **-Xms<value>**

Sets the initial Java heap size. You can also use **-Xmo**. The minimum size is 8 KB.

If scavenger is enabled,  $\text{-Xms} \geq \text{-Xmn} + \text{-Xmo}$ )

If scavenger is disabled,  $\text{-Xms} \geq \text{-Xmo}$ )

### **-Xmx<value>**

Sets the maximum memory size ( $\text{Xmx} \geq \text{Xms}$ )

Examples of the use of **-Xms** and **-Xmx** are:

#### **-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.

## Garbage Collector command-line options

### **-Xnoclassgc**

Disables class garbage collection. This 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**.

### **-Xnocompactexplicitgc**

Never runs compaction on system garbage collections. Compaction takes place on global garbage collections if you specify **-Xcompactgc** or if compaction triggers are met.

### **-Xnocompactgc**

Disables compaction on all garbage collections (system or global).

### **-Xnoloa**

Prevents allocation of a large object area; all objects will be allocated in the SOA. See also **-Xloa**.

### **-Xnopartialcompactgc**

Disables incremental compaction. See also **-Xpartialcompactgc**. By default, this option is not set, so all compactions are full.

### **-Xpartialcompactgc**

Enables incremental compaction. See also **-Xnopartialcompactgc**.

### **-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 runtime. By default, this option is set to the same value as **-Xmx**.

### **-Xsoftrefthreshold<number>**

Sets the number of GCs after which a soft reference will be 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 will be 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 its 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, 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. Similar to **terse**.

#### **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 in 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 a reference object is enqueued for finalisation, showing the reference type, reference address, and referent address.

### **scavenger**

Prints extra information following 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 space is linearly walked to achieve this.

### **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:<path to file><filename>**

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 will redirect the output to stderr.

### **-Xverbosegclog:<path to file><filename, X, Y>**

X and Y are integers. This option works similarly to **-Xverbosegclog:<path to file><filename>**, but, in addition, the verbose:gc output is redirected to X files, each containing verbose:gc output from Y GC cycles.

## **Garbage Collector command-line options**

---

## Appendix E. Default settings for the JVM

This appendix shows the default settings that the JVM uses; that is, how the JVM operates if you do not apply any changes to its environment. The tables show the JVM operation and the default setting.

The last column shows how the operation setting is affected and is set as follows:

- **e** – setting controlled by environment variable only
- **c** – setting controlled by command-line parameter only
- **ec** – setting controlled by both (command-line always takes precedence) All the settings are described elsewhere in this document. These tables are only a quick reference to the JVM vanilla state

For default GC settings, see Chapter 2, “Understanding the Garbage Collector,” on page 7.

*Table 22. Cross platform defaults*

JVM setting	Default	Setting affected by
Javadumps	Enabled	ec
Javadumps on out of memory	Enabled	ec
Heapsdumps	Disabled	ec
Heapsdumps on out of memory	Enabled	ec
Sysdumps	Enabled	ec
Where dump files appear	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 signalling	Disabled	c
Signal handler chaining	Enabled	c
Classpath	Not set	ec
Accessibility support	Enabled	e
JIT	Enabled	ec
AOT compiler	Enabled	c
JIT debug options	Disabled	c
Java2D max size of fonts with algorithmic bold	14 point	e
Java2D use rendered bitmaps in scalable fonts	Enabled	e
Java2D freetype font rasterizing	Enabled	e
Java2D use AWT fonts	Disabled	e

## Default settings for the JVM

Table 23. Platform-specific defaults

JVM setting	AIX	Linux	Windows	z/OS	Setting affected by
Default locale	None	None	N/A	None	e
Time to wait before starting plug-in	N/A	Zero	N/A	N/A	e
Temporary directory	/tmp	/tmp	\tmp	/tmp	e
Plug-in redirection	None	None	N/A	None	e
IM switching	Disabled	Disabled	N/A	Disabled	e
IM modifiers	Disabled	Disabled	N/A	Disabled	e
Thread model	N/A	N/A	N/A	Native	e
Initial stack size for Java Threads 32-bit Use: <code>-Xiss&lt;size&gt;</code>	2 KB	2 KB	2 KB	2 KB	c
Maximum stack size for Java Threads 32-bit 32-bit Use: <code>-Xss&lt;size&gt;</code>	256 KB	256 KB	256 KB	256 KB	c
Stack size for OS Threads 32-bit Use <code>-Xmso&lt;size&gt;</code>	256 KB	256 KB	32 KB	256 KB	c
Initial stack size for Java Threads 64-bit Use: <code>-Xiss&lt;size&gt;</code>	2 KB	2 KB	2 KB	2 KB	c
Maximum stack size for Java Threads 64-bit 64-bit Use: <code>-Xss&lt;size&gt;</code>	512 KB	512 KB	512 KB	512 KB	c
Stack size for OS Threads 64-bit Use <code>-Xmso&lt;size&gt;</code>	256 KB	256 KB	256 KB	256 KB	c
Initial heap size Use <code>-Xms&lt;size&gt;</code>	4 MB	4 MB	4 MB	1 MB	c
Maximum Java heap size Use <code>-Xmx&lt;size&gt;</code>	64 MB	Half the real storage with a minimum of 16 MB and a maximum of 512 MB -1	Half the real storage with a minimum of 16 MB and a maximum of 2 GB -1	64 MB	c

---

## Appendix F. 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:

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106-0032, 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.

## Notices

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.

### 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

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, or other countries, or both.

AIX, DB2, IBM, OS/390, S/390, WebSphere, z/OS

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

## **Trademarks**

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

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=1 171  
-verbose:dynload 265  
-verbose:gc (garbage collection) 248  
-Xalwaysclassgc  
    garbage collection 258  
-Xclassgc  
    garbage collection 258  
-Xcompactexplicitgc  
    garbage collection 259  
-Xcompactgc  
    garbage collection 259  
-Xconcurrentbackground<number>  
    garbage collection 258  
-Xconcurrentlevel<number>  
    garbage collection 258  
-Xcommeter:<soa | loa | dynamic>  
    garbage collection 259  
-Xdisableexcessivegc  
    garbage collection 260  
-Xdisableexplicitgc  
    garbage collection 258  
-Xdisablestringconstantgc  
    garbage collection 259  
-Xenableexcessivegc  
    garbage collection 260  
-Xenablestringconstantgc  
    garbage collection 259  
-Xgcthreads  
    garbage collection 258  
-Xgworkpackets<number>  
    garbage collection 258  
-Xloainitial<number>  
    garbage collection 259  
-Xloamaximum<number>  
    garbage collection 259  
-Xlp  
    garbage collection 259  
-Xnaclassgc  
    garbage collection 258  
-Xnocompactexplicitgc  
    garbage collection 259  
-Xnocompactgc  
    garbage collection 259  
-Xnoloa  
    garbage collection 259  
-Xnopartialcompactgc  
    garbage collection 259  
-Xpartialcompactgc  
    garbage collection 259  
-Xsoftrethreshold<number>  
    garbage collection 260  
-Xtgc:backtrace  
    garbage collection 260  
-Xtgc:compaction  
    garbage collection 260  
-Xtgc:concurrent  
    garbage collection 261  
-Xtgc:dump  
    garbage collection 261

-Xtgc:freelist  
    garbage collection 261  
-Xtgc:parallel  
    garbage collection 262  
-Xtgc:references  
    garbage collection 262  
-Xtgc:scavenger  
    garbage collection 262  
-Xtgc:terse  
    garbage collection 263  
-Xtrace 171  
.dat files 299  
.nix platforms  
    font utilities 184  
-Xdump 226  
see also jdumpview 225

## Numerics

32- and 64-bit JVMs  
    AIX 112  
32-bit AIX Virtual Memory Model, AIX 112  
64-bit AIX Virtual Memory Model, AIX 113

## A

about this book xiii  
Addr Range, AIX segment type 105  
agent, JVMRI  
    building  
        Linux 308  
        Windows 307  
        z/OS 308  
    launching 307  
    writing 305  
AIX  
    available disk space 99  
    checking environment 97  
    crashes 107  
    debugging commands 99  
        archon 103  
        band 103  
        bindprocessor 99  
        bindprocessor -q 99  
        bootinfo 99  
        cmd 102  
        cp 103  
        dbx 99  
        dbx Plug-in 107  
        Esid 104  
        f 103  
        iostat 100  
        lsattr 100  
        netpmmon 100  
        netstat 101  
        nmon 102  
        pid 102  
        ppid 102  
        pri 103  
        ps 102

AIX (*continued*)

  debugging commands (*continued*)

- sar 103
- sc 103
- st 103
- stime 102
- svmon 104
- tat 103
- tid 103
- time 102
- topas 105
- tprof 105
- trace 106
- truss 106
- tty 102
- Type 104
- uid 102
- user 103
- vmstat 106
- Vsid 104

  debugging hangs 109

- AIX deadlocks 109

- investigating busy hangs 110

- poor performance 112

  debugging memory leaks

- 32- and 64-bit JVMs 112
- 32-bit AIX Virtual Memory Model 112
- 64-bit AIX Virtual Memory Model 113
- changing the Memory Model (32-bit JVM) 113
- fragmentation problems 118
- Java heap exhaustion 118
- Java or native heap exhausted 117
- Java2 32-Bit JVM default memory models 115
- monitoring the Java heap 117
- monitoring the native heap 115
- native and Java heaps 114
- native heap exhaustion 118
- native heap usage 116
- receiving OutOfMemory errors 117
- submitting a bug report 119

  debugging performance problems 119

- application profiling 125
- collecting data from a fault condition 125

  CPU bottlenecks 120

- finding the bottleneck 120

- getting AIX technical support 126

  I/O bottlenecks 125

  JIT compilation 125

  JVM heap sizing 125

- memory bottlenecks 124

  debugging techniques 99

  diagnosing crashes 108

- documents to gather 108

- interpreting the stack trace 108

  enabling full AIX core files 98

  Heapdumps 99

  Java Virtual Machine settings 98

  Javadumps 99

  JVM dump initiation 220

  MALLOCTYPE=watson 117

  operating system settings 98

  problem determination 97

  setting up and checking AIX environment 97

  stack trace 108

  technical support 126

  understanding memory usage 112

  allocation failures 250

analyzing deadlocks, Windows 147

  API calls, JVMRI 308

- CreateThread 308

- DumpDeregister 309

- DumpRegister 309

- dynamic verbosegc 309

- GenerateHeapdump 310

- GenerateJavacore 310

- GetComponentDataArea 310

- GetRasInfo 310

- InitiateSystemDump 311

- InjectOutOfMemory 311

- InjectSigsegv 311

- NotifySignal 311

- ReleaseRasInfo 312

- RunDumpRoutine 312

- SetOutOfMemoryHook 312

- TraceDeregister 312

- TraceRegister 313

- TraceResume 313

- TraceResumeThis 313

- TraceSet 313

- TraceSnap 314

- TraceSuspend 314

- TraceSuspendThis 314

application profiling, AIX 125

  application profiling, Linux 138

  application profiling, Windows 150

  application profiling, z/OS 164

  application stack 4

  application trace 300

- activating and deactivating tracepoints 303

- example 302

- printf specifiers 303

- registering 300

- suspend or resume 304

- trace api 303

- trace buffer snapshot 304

- tracepoints 301

- using at runtime 303

archon, AIX 103

## B

BAD\_OPERATION 172

BAD\_PARAM 172

band, AIX 103

before you read this book xiii

bidirectional GIOP, ORB limitation 171

bindprocessor -q, AIX 99

bindprocessor, AIX 99

bootinfo , AIX 99

bottlenecks, AIX

- CPU 120

- finding 120

- I/O 125

- memory 124

bottlenecks, Windows

- finding 149

bottlenecks, z/OS

- finding 163

buffers

- snapping 285

- trace 285

bug report

- garbage collection 22

busy hangs, AIX (investigating) 110

# C

cache allocation (garbage collection) 10

cache housekeeping

  shared classes 271

cache naming

  shared classes 270

cache performance

  shared classes 271

cache problems

  shared classes 279, 282

categorizing problems 189

changing the Memory Model (32-bit JVM), AIX 113

checking and setting up environment, Windows 143, 144

checklist for problem submission 83

  before you submit 83

  data to include 83

  factors that affect JVM performance 84

  performance problem questions 85

  test cases 84

class GC

  shared classes 273

class loader 6

  how to write a custom class loader 31

  name spaces and the runtime package 30

  parent-delegation model 29

  understanding 29

  why write your own class loader? 30

class-loader diagnostics 265

  command-line options 265

  loading from native code 266

  runtime 265

client side interception points, ORB 58

  receive\_exception (receiving reply) 58

  receive\_other (receiving reply) 58

  receive\_reply (receiving reply) 58

  send\_poll (sending request) 58

  send\_request (sending request) 58

client side, ORB 49

  getting hold of the remote object 51

    bootstrap process 51

  identifying 177

  ORB initialization 50

  remote method invocation 52

    delegation 52

    servant 52

  stub creation 49

clnt , AIX segment type 105

cmd, AIX 102

codes, minor (CORBA) 337

collecting data from a fault condition

  AIX 125

  Linux 138, 139

    core files 138

      determining the operating environment 139

    proc file system 139

    producing Javadumps 138

    sending information to Java Support 139

    strace, ltrace, and mtrace 139

    using system logs 138

  Windows 150

  z/OS 164

com.ibm.CORBA.AcceptTimeout 46

com.ibm.CORBA.AllowUserInterrupt 46

com.ibm.CORBA.BootstrapHost 46

com.ibm.CORBA.BootstrapPort 46

com.ibm.CORBA.BufferSize 46

com.ibm.CORBA.CommTrace 171

com.ibm.CORBA.ConnectionMultiplicity 46

com.ibm.CORBA.ConnectTimeout 46

com.ibm.CORBA.Debug 171

com.ibm.CORBA.Debug.Output 171

com.ibm.CORBA.enableLocateRequest 47

com.ibm.CORBA.FragmentSize 47

com.ibm.CORBA.FragmentTimeout 47

com.ibm.CORBA.GIOPAddressingDisposition 47

com.ibm.CORBA.InitialReferencesURL 47

com.ibm.CORBA.ListenerPort 47

com.ibm.CORBA.LocalHost 47

com.ibm.CORBA.LocateRequestTimeout 47, 178

com.ibm.CORBA.MaxOpenConnections 48

com.ibm.CORBA.MinOpenConnections 48

com.ibm.CORBA.NoLocalInterceptors 48

com.ibm.CORBA.ORBCharEncoding 48

com.ibm.CORBA.ORBWCharDefault 48

com.ibm.CORBA.RequestTimeout 48, 178

com.ibm.CORBA.SendingContextRunTimeSupported 48

com.ibm.CORBA.SendVersionIdentifier 48

com.ibm.CORBA.ServerSocketQueueDepth 48

com.ibm.CORBA.ShortExceptionDetails 48

com.ibm.tools.rmic.iop.Debug 48

com.ibm.tools.rmic.iop.SkipImports 48

comm trace , ORB 176

COMM\_FAILURE 172

command-line options 361

  class-loader 265

  garbage collector 370

  general 361

  JIT 369

  nonstandard 364

  system property 363

commands, (IPCS), z/OS 152

compaction phase (garbage collection) 8

  detailed description 15

compilation failures, JIT 245

COMPLETED\_MAYBE 173

COMPLETED\_NO 173

COMPLETED\_YES 173

completion status, ORB 173

concurrent access

  shared classes 272

concurrent mark (garbage collection) 13

connection handlers 76

console dumps 197

control flow optimizations (JIT) 36

conventions and terminology in book xiv

CORBA 39

  client side interception points 58

  receive\_exception (receiving reply) 58

  receive\_other (receiving reply) 58

  receive\_reply (receiving reply) 58

  send\_poll (sending request) 58

  send\_request (sending request) 58

  examples 40

  fragmentation 57

  further reading 40

  interfaces 40

  interoperable naming service (INS) 60

  Java IDL or RMI-IIOP, choosing 40

  minor codes 337

  portable interceptors 58

  portable object adapter 55

  remote object implementation (or servant) 41

  RMI and RMI-IIOP 39

  RMI-IIOP limitations 40

CORBA (*continued*)  
server code 42  
differences between RMI (JRMP) and RMI-IIOP 45  
summary of differences in client development 45  
summary of differences in server development 45  
server side interception points 58  
receive\_request (receiving request) 58  
receive\_request\_service\_contexts (receiving request) 58  
send\_exception (sending reply) 58  
send\_other (sending reply) 58  
send\_reply (sending reply) 58  
stubs and ties generation 41  
core dump 217  
defaults 217  
environment variables 218  
overview 217  
platform-specific variations 219  
core dumps  
Linux 129  
core files  
Linux 127  
core files, Linux 138  
cp, AIX 103  
CPU bottlenecks, AIX 120  
CPU usage, Linux 136  
crashes  
AIX 107  
Linux 134  
Windows 146  
z/OS 153  
documents to gather 153  
failing function 154  
crashes, diagnosing  
Windows  
sending data to IBM 147  
CreateThread, JVMRI 308  
cross-platform tools  
dump formatter 191  
Heapdump 191  
JPDA tools 192  
JVMPi tools 192  
JVMRI 193  
JVMTI 192  
trace formatting 192

## D

data submission with problem report 89  
sending files to IBM support 89  
data to be collected, ORB 181  
DATA\_CONVERSION 172  
dbx Plug-in, AIX 107  
dbx, AIX 99  
deadlocked process, z/OS 160  
deadlocks 109, 207  
deadlocks, Windows  
debugging 147  
debug properties, ORB 171  
com.ibm.CORBA.CommTrace 171  
com.ibm.CORBA.Debug 171  
com.ibm.CORBA.Debug.Output 171  
debugging commands  
AIX 99  
bindprocessor -q 99  
bootinfo 99  
dbx 99  
dbx Plug-in 107

debugging commands (*continued*)  
AIX (*continued*)  
iostat 100  
lsattr 100  
netpmmon 100  
netstat 101  
nmon 102  
sar 103  
topas 105  
tprof 105  
trace 106  
truss 106  
vmstat 106  
Linux 131  
debugging hangs, AIX 109  
AIX deadlocks 109  
investigating busy hangs 110  
poor performance 112  
debugging hangs, Windows 147  
debugging memory leaks, AIX  
32- and 64-bit JVMs 112  
32-bit AIX Virtual Memory Model 112  
64-bit AIX Virtual Memory Model 113  
changing the Memory Model (32-bit JVM) 113  
fragmentation problems 118  
Java heap exhaustion 118  
Java or native heap exhausted 117  
Java2 32-Bit JVM default memory models 115  
monitoring the Java heap 117  
monitoring the native heap 115  
native and Java heaps 114  
native heap exhaustion 118  
native heap usage 116  
receiving OutOfMemory errors 117  
submitting a bug report 119  
debugging memory leaks, Windows  
memory model 148  
tracing leaks 148  
debugging performance problem, AIX  
application profiling 125  
collecting data from a fault condition 125  
CPU bottlenecks 120  
finding the bottleneck 120  
getting AIX technical support 126  
I/O bottlenecks 125  
JIT compilation 125  
JVM heap sizing 125  
memory bottlenecks 124  
debugging performance problem, Linux  
JIT compilation 138  
JVM heap sizing 138  
debugging performance problem, Linuxs  
application profiling 138  
debugging performance problem, Windows  
application profiling 150  
finding the bottleneck 149  
JIT compilation 150  
JVM heap sizing 150  
systems resource usage 149  
debugging performance problem, z/OS  
application profiling 164  
finding the bottleneck 163  
JIT compilation 164  
JVM heap sizing 163  
systems resource usage 163  
debugging performance problems, AIX 119

debugging performance problems, Linux  
  CPU usage 136  
  finding the bottleneck 136  
  memory usage 137  
  network problems 137

debugging performance problems, Windows 149

debugging techniques, AIX 99  
  bindprocessor -q 99  
  bootinfo 99  
  dbx 99  
  dbx Plug-in 107  
  debugging commands 99  
  iostat 100  
  lsattr 100  
  netpmmon 100  
  netstat 101  
  nmon 102  
  sar 103  
  starting Heapdumps 99  
  starting Javadumps 99  
  topas 105  
  tprof 105  
  trace 106  
  truss 106  
  vmstat 106

debugging techniques, Linux  
  ps command 129  
  vmstat command  
    processes section 130

debugging techniques, Windows 145  
  Dump Formatter 145  
  Heapdumps 145  
  Javadumps 145

default memory models, Java2 32-Bit JVM (AIX) 115

default settings, JVM 377

defaults  
  core dump 217

delegation, ORB client side 52

deploying shared classes 270

deprecated Sun properties 49

description string, ORB 174

Description, AIX segment type 105

determining the operating environment, Linux 139

df command, Linux 139

diagnosing crashes, AIX 108  
  documents to gather 108  
  interpreting the stack trace 108

diagnostics component 5

diagnostics options, JVM environment 343

diagnostics, class loader  
  loading from native code 266  
  runtime 265

diagnostics, class-loader 265  
  command-line options 265

diagnostics, overview 189  
  categorizing problems 189  
  cross-platform tools 191  
    dump formatter 191  
    Heapdump 191  
    JPDA tools 192  
    JVMPI tools 192  
    JVMRI 193  
    JVMTI 192  
    trace formatting 192  
  platforms 189

differences between RMI (JRMP) and RMI-IIOP, ORB 45

disabling the JIT 243

Distributed Garbage Collection (DGC)  
  RMI 76

documents to gather  
  AIX 108

DTFJ  
  counting threads example 331  
  diagnostics 327  
  interface diagram 329  
  overview 327  
  working with a dump 327

dump  
  AIX and Linux 220  
  core 217  
    defaults 217  
    environment variables 218  
    overview 217  
    platform-specific variations 219  
  Windows 220  
  z/OS 219

dump agents  
  console dumps 197  
  default 199  
  default settings 200  
  examples 197  
  filters 201  
  heapdumps 198  
  help options 195  
  Java dumps 198  
  removing 201  
  snap traces 199  
  system dumps 197  
  tool option 198  
  triggering 197  
  types 197  
  using 195

dump extraction  
  Windows 144

dump extractor  
  Linux 129

dump formatter 225  
  analyzing dumps 232  
  cross-platform tools 191  
  example session 232  
  problems to tackle with 226

dump, generated (Javadump) 203

DumpDeregister, JVMRI 309

DumpRegister, JVMRI 309

dumps, setting up (z/OS) 152

dynamic updates  
  shared classes 275

dynamic verbosegc, JVMRI 309

## E

enabling full AIX core files 98

environment  
  checking on AIX 97  
  displaying current 339  
  JVM settings 339  
    deprecated JIT options 341  
    diagnostics options 343  
    general options 340  
    Javadump and Heapdump options 341  
  setting up and checking on Windows 143, 144  
  setting up on Windows  
    dump extraction 144  
    native tools 144

environment variables 339  
core dump 218  
heaps 215  
javadumps 210  
separating values in a list 339  
setting 339  
z/OS 151, 343

environment, determining  
Linux 139  
df command 139  
free command 139  
lsof command 139  
ps-ef command 139  
top command 139  
uname -a command 139  
vmstat command 139

error message IDs  
z/OS 153

errors (OutOfMemory), receiving (AIX) 117

Esid, AIX 104

example of real method trace 223

examples of method trace 222

exceptions, JNI 69

exceptions, ORB 172  
completion status and minor codes 173  
nested 175  
system 172  
BAD\_OPERATION 172  
BAD\_PARAM 172  
COMM\_FAILURE 172  
DATA\_CONVERSION 172  
MARSHAL 172  
NO\_IMPLEMENT 172  
UNKNOWN 172  
user 172

exhaustion of Java heap, AIX 118

exhaustion of native heap, AIX 118

external trace, JVMRI 316

## F

f, AIX 103  
failing function, z/OS 154  
failing method, JIT 244  
fault condition in AIX  
collecting data from 125  
file header, Javdump 205  
finalizers 248  
finding classes  
shared classes 276  
finding the bottleneck, Linux 136  
first steps in problem determination 95  
floating stacks limitations, Linux 140  
font limitations, Linux 140  
fonts, NLS 183  
common problems 184  
installed 183  
properties 183  
utilities 184  
.nix platforms 184  
Windows systems 184

formatting, JVMRI 316

fragmentation  
AIX 118  
ORB 57, 170

free command, Linux 139

frequently asked questions  
garbage collection 25  
JIT 37  
functions (table), JVMRI 308

**G**

garbage collection 8  
advanced diagnostics 257  
-Xalwaysclassgc 258  
-Xclassgc 258  
-Xcompactexplicitgc 259  
-Xcompactgc 259  
-Xconcurrentbackground<number> 258  
-Xconcurrentlevel<number> 258  
-Xcometer:<soa | loa | dynamic> 259  
-Xdisableexcessivegc 260  
-Xdisableexplicitgc 258  
-Xdisablestringconstantgc 259  
-Xenableexcessivegc 260  
-Xenablestringconstantgc 259  
-Xgcthreads 258  
-Xgcworkpackets<number> 258  
-Xloainit<number> 259  
-Xloamaximum<number> 259  
-Xlp 259  
-Xnklassgc 258  
-Xnocompactexplicitgc 259  
-Xnocompactgc 259  
-Xnoloa 259  
-Xnopartialcompactgc 259  
-Xpartialcompactgc 259  
-Xsoftrefthreshold<number> 260  
-Xtgc:backtrace 260  
-Xtgc:compaction 260  
-Xtgc:concurrent 261  
-Xtgc:dump 261  
-Xtgc:freelist 261  
-Xtgc:parallel 262  
-Xtgc:references 262  
-Xtgc:scavenger 262  
-Xtgc:terse 263  
TGC tracing 260

allocation failures 250  
allocation failures during concurrent mark 254  
basic diagnostics (verbose:gc) 248  
cache allocation 10  
coexisting with the Garbage Collector 22  
bug reports 22  
finalizers 23  
finalizers and the garbage collection contract 23  
finalizers, summary 24  
how finalizers are run 24  
manual invocation 24  
nature of finalizers 23  
summary 24  
thread local heap 22

command-line options 370  
common causes of perceived leaks 247  
hash tables 248  
JNI references 248  
listeners 247  
objects with finalizers 248  
premature expectation 248  
static data 248

compaction phase 8  
detailed description 15

garbage collection (*continued*)  
concurrent 253  
concurrent kickoff 253  
concurrent mark 13  
concurrent sweep completed 253  
detailed description 11  
fine tuning options 21  
frequently asked questions 25  
Generational Garbage Collector 18  
global collections 251  
heap and native memory use by the JVM 263  
    large native objects 264  
heap expansion 17  
heap lock allocation 10  
heap shrinkage 17  
heap size 8  
    problems 9  
how does it work? 247  
how to do heap sizing 20  
initial and maximum heap sizes 20  
interaction with applications 21  
interaction with JNI 64  
    global references 65  
    object references 64  
JNI weak reference 16  
Large Object Area 10  
mark phase 8  
    detailed description 11  
    mark stack overflow 12  
    parallel mark 12  
memory allocation 9  
native code 263  
nursery allocation failures 250  
object allocation 7  
output from a System.gc() 249  
overview 7  
parallel bitwise sweep 14  
phantom reference 16  
reachable objects 8  
reference objects 15  
scavenger collections 252  
soft reference 16  
sweep phase 8  
    detailed description 14  
System.gc() calls during concurrent mark 256  
tenure age 19  
tenured allocation failures 251  
tilt ratio 19  
understanding the Garbage Collector 7  
using verbose:gc 21  
verbose, heap information 215  
weak reference 16  
GenerateHeapdump, JVMRI 310  
GenerateJavacore, JVMRI 310  
generating a user dump file in a hang condition,  
    Windows 144  
generation of a Heapdump  
    location 214  
Generational Garbage Collector  
    sizing, garbage collection 18  
    tenure age 19  
    tilt ratio 19  
GetComponentDataArea, JVMRI 310  
GetRasInfo, JVMRI 310  
getting a dump from a hung JVM, Windows 147  
getting AIX technical support 126  
getting files from IBM support 90

glibc limitations, Linux 140  
global optimizations (JIT) 37  
global references (JNI) 69  
    capacity 69  
growing classpaths  
    shared classes 272

## H

hanging, ORB 178  
com.ibm.CORBA.LocateRequestTimeout 178  
com.ibm.CORBA.RequestTimeout 178

hangs  
    AIX  
        investigating busy hangs 110  
    Windows  
        debugging 147  
    z/OS 160  
        bad performance 161

hangs, debugging  
    AIX 109  
        AIX deadlocks 109  
        poor performance 112

hash tables 248

heap  
    expansion 17  
    lock allocation 10  
    shrinkage 17  
    size, garbage collection 8  
        problems 9  
    sizing, garbage collection 20

heap (Java) exhaustion, AIX 118

heap and native memory use by the JVM  
    garbage collection  
        large native objects 264

heap, verbose GC 215

Heapdump 213  
    AIX, starting 99  
    cross-platform tools 191  
    enabling 213  
    environment variables 215  
    Linux, starting 129  
    location of 214  
    phd format 213  
    previous releases 213  
    summary 213  
    using jdumpview 214  
    Windows  
        starting 145

heapdumps 198

heaps, native and Java  
    AIX 114

Hewlett-Packard  
    problem determination 167

how to read this book xiii

HPROF Profiler 319  
    options 319  
    output file 321

hung JVM  
    getting a dump from  
        Windows 147

## I

I/O bottlenecks, AIX 125

initialization problems  
  shared classes 280  
InitiateSystemDump, JVMRI 311  
InjectOutOfMemory, JVMRI 311  
InjectSigsegv, JVMRI 311  
Inlining (JIT) 36  
INS, ORB  
  *See* interoperable naming service  
interceptors (portable), ORB 58  
Interface Definition Language (IDL) 40  
interoperable naming service (INS), ORB 60  
interpreter 6  
interpreting the stack trace, AIX 108  
Inuse, AIX segment type 105  
investigating busy hangs, AIX 110  
iostat, AIX 100  
IPCS commands, z/OS 152

## J

jar and zip files  
  shared classes 271  
Java dumps 198  
Java duty manager 81  
Java heap, AIX 114  
  exhaustion 117, 118  
  monitoring 117  
Java Helper API  
  shared classes 277  
Java Native Interface  
  *see* JNI 63  
Java or native heap exhausted, AIX 117  
Java service  
  overview 81  
    IBM service 81  
    Java duty manager 81  
  submitting problem report to IBM 81  
JAVA\_DUMP\_OPTS 343  
  default dump agents 199  
  JVMRI 311  
  parsing 218  
  setting up dumps 152  
JAVA\_LOCAL\_TIME 343  
JAVA\_TDUMP\_PATTERN=string 343  
JAVA\_THREAD\_MODEL 343  
Java2 32-Bit JVM default memory models, AIX 115  
Javadump 203  
  enabling 203  
  environment variables 210  
  file header, gpinfo 205  
  file header, title 205  
  interpreting 204  
  Linux 128  
  Linux, producing 138  
  location of generated dump 203  
  locks, monitors, and deadlocks (LOCKS) 207  
  storage management 207  
  system properties 206  
  tags 205  
  triggering 204  
  Windows 145  
Javadumps  
  AIX 99  
javasharedresources  
  shared classes 281  
jdmpview 225  
  commands 227

jdmpview (*continued*)  
  general 227  
  heapdump 231  
  memory analysis 229  
  trace 232  
  working with classes 230  
  working with objects 230  
example session 232  
jextract 225  
  overview 225  
jextract 225  
JIT  
  command-line options 369  
  compilation failures, identifying 245  
  control flow optimizations 36  
  disabling 243  
  frequently asked questions 37  
  global optimizations 37  
  how the JIT optimizes code 36  
  Inlining 36  
  JVM environment options 341  
  local optimizations 36  
  locating the failing method 244  
  native code generation 37  
  ORB-connected problem 170  
  overview 35  
  problem determination 243  
  selectively disabling 243  
  short-running applications 246  
  understanding 35  
JIT compilation  
  AIX 125  
  Linux 138  
  Windows 150  
  z/OS 164  
JNI 63  
  checklist 73  
  copying and pinning 65  
  debugging 72  
  exceptions 69  
  garbage collection 16  
  generic use of isCopy and mode flags 71  
  global references 69  
  interaction with Garbage Collector 64  
    global references 65  
    object references 64  
  isCopy flag 69  
  local references 66  
  mode flag 70  
  problem determination 72  
  references for garbage collection 248  
  synchronization 71  
  understanding 63  
  weak reference 16  
JPDA tools, cross-platform tools 192  
JVM  
  API 5  
  application stack 4  
  building blocks 3  
  class loader 6  
  diagnostics component 5  
  environment settings 339  
    deprecated JIT options 341  
    diagnostics options 343  
    general options 340  
  Javadump and Heapdump options 341  
  interpreter 6

JVM (*continued*)  
    memory management 5  
    platform port layer 6  
    subcomponents 4  
    trace formatting 192  
JVM dump initiation  
    AIX and Linux 220  
    Windows 220  
    z/OS 219  
JVM heap sizing  
    AIX 125  
    Linux 138  
    Windows 150  
    z/OS 163  
JVMPi  
    cross-platform tools 192  
JVMRI 305  
    agent design 308  
    API calls 308  
        CreateThread 308  
        DumpDeregister 309  
        DumpRegister 309  
        dynamic verbosegc 309  
        GenerateHeapdump 310  
        GenerateJavacore 310  
        GetComponentDataArea 310  
        GetRasInfo 310  
        InitiateSystemDump 311  
        InjectOutOfMemory 311  
        InjectSigsegv 311  
        NotifySignal 311  
        ReleaseRasInfo 312  
        RunDumpRoutine 312  
        SetOutOfMemoryHook 312  
        TraceDeregister 312  
        TraceRegister 313  
        TraceResume 313  
        TraceResumeThis 313  
        TraceSet 313  
        TraceSnap 314  
        TraceSuspend 314  
        TraceSuspendThis 314  
building the agent  
    Linux 308  
    Windows 307  
    z/OS 308  
changing trace options 307  
cross-platform tools 193  
external trace 316  
    formatting 316  
    functions (table) 308  
launching the agent 307  
RasInfo  
    request types 315  
    structure 314  
registering a trace listener 306  
writing an agent 305  
JVMTI  
    cross-platform tools 192  
    diagnostics 325

## K

kernel, AIX segment type 105  
known limitations, Linux 140  
    floating stacks limitations 140  
    font limitations 140

known limitations, Linux (*continued*)  
    glibc limitations 140  
    threads as processes 140  
  
**L**  
large native objects  
    heap and native memory use by the JVM  
        garbage collection 264  
Large Object Area (garbage collection) 10  
LE HEAP, z/OS 161  
LE settings, z/OS 151  
limitations, Linux 140  
    floating stacks limitations 140  
    font limitations 140  
    glibc limitations 140  
    threads as processes 140  
Linux  
    checking the system environment 134  
    collecting data from a fault condition 138, 139  
        core files 138  
        determining the operating environment 139  
        proc file system 139  
        producing Javadumps 138  
        sending information to Java Support 139  
        strace, ltrace, and mtrace 139  
        using system logs 138  
    core files 127  
    crashes, diagnosing 134  
    debugging commands 131  
        gdb 132  
        ltrace tool 131  
        mtrace tool 131  
        ps 131  
        strace tool 131  
        tracing 131  
    debugging hangs 135  
    debugging memory leaks 135  
    debugging performance problems 136  
        application profiling 138  
        CPU usage 136  
        finding the bottleneck 136  
        JIT compilation 138  
        JVM heap sizing 138  
        memory usage 137  
        network problems 137  
    debugging techniques 128  
    finding out about the Java environment 135  
gathering process information 134  
JVM dump initiation 220  
known limitations 140  
    floating stacks limitations 140  
    font limitations 140  
    glibc limitations 140  
    threads as processes 140  
ltrace 139  
mtrace 139  
nm command 129  
objdump command 129  
problem determination 127  
ps command 129  
setting up and checking the environment 127  
starting heapdumps 129  
starting Javadumps 128  
strace 139  
threading libraries 128  
top command 130

**Linux** (*continued*)  
 tracing 131  
 using core dumps 129  
 using system logs 129  
 using the dump extractor 129  
 vmstat command 130  
     io section 130  
     memory section 130  
     processes section 130  
     swap section 130  
     system section 130  
 working directory 127  
 listeners 247  
 local optimizations (JIT) 36  
 local references (JNI) 66  
     capacity 68  
     manually handling 68  
     scope 66  
     summary 68  
 locating the failing method, JIT 244  
 location of generated Heapdump 214  
 locks, monitors, and deadlocks (LOCKS), Javadump 207  
 long classpaths  
     shared classes 272  
 looping process, z/OS 160  
 lsattr, AIX 100  
 lsof command, Linux 139  
 ltrace, Linux 139

## M

maintenance, z/OS 151  
 MALLOCTYPE=watson 117  
 mark phase (garbage collection) 8  
     concurrent mark 13  
     detailed description 11  
     parallel mark 12  
 MARSHAL 172  
 memory allocation 9  
     cache allocation 10  
     heap lock allocation 10  
     Large Object Area 10  
 memory bottlenecks, AIX 124  
 memory leaks  
     Windows  
         classifying 148  
         debugging 148  
     z/OS 161  
         LE HEAP 161  
         OutOfMemoryErrors 162  
         virtual storage 161  
 memory leaks, debugging  
     AIX  
         32- and 64-bit JVMs 112  
         32-bit AIX Virtual Memory Model 112  
         64-bit AIX Virtual Memory Model 113  
         changing the Memory Model (32-bit JVM) 113  
         Java heap exhaustion 118  
         Java or native heap exhausted 117  
         Java2 32-Bit JVM default memory models 115  
         monitoring the Java heap 117  
         monitoring the native heap 115  
         native and Java heaps 114  
         native heap exhaustion 118  
         native heap usage 116  
         receiving OutOfMemory errors 117

memory leaks, Windows  
     tracing 148  
 memory management 5  
 Memory Model (32-bit JVM), changing, AIX 113  
 memory model, Windows 148  
 memory models, Java2 32-Bit JVM default (AIX) 115  
 memory usage, Linux 137  
 memory usage, understanding  
     AIX 112  
 message trace, ORB 175  
 messages 345  
     dump 345  
     j9vm 347  
     shrc 349  
 method trace 221  
     advanced options 222  
     examples 222  
     real example 223  
     running with 221  
     where output appears 222  
 minor codes, CORBA 337  
 minor codes, ORB 173  
 mmap, AIX segment type 105  
 modification contexts  
     shared classes 274  
 monitoring the Java heap, AIX 117  
 monitoring the native heap, AIX 115  
 monitors, Javadump 207  
 mtrace, Linux 139  
 MustGather  
     collecting the correct data to solve problems 83

## N

native code  
     garbage collection 263  
 native code generation (JIT) 37  
 native heap, AIX 114  
     exhaustion 117, 118  
     monitoring 115  
     usage 116  
 native tools  
     Windows 144  
 nested exceptions, ORB 175  
 netpmmon, AIX 100  
 netstat, AIX 101  
 network problems, Linux 137  
 NLS  
     font properties 183  
     fonts 183  
     installed fonts 183  
     problem determination 183  
 nmon, AIX 102  
 NO\_IMPLEMENT 172  
 NotifySignal, JVMRI 311

## O

object allocation 7  
 objects  
     reachable 8  
 objects with finalizers 248  
 objects, reference (garbage collection)  
     Garbage Collector interaction with JNI 64  
 options  
     command-line 361

options (*continued*)  
     general 361  
     nonstandard 364  
     system property 363

JVM environment  
     deprecated JIT 341  
     diagnostics 343  
     general 340  
     method trace, advanced 222

options that control tracepoint selection 287

options that indirectly affect tracepoint selection 288

ORB 39  
     bidirectional GIOP limitation 171  
     choosing Java IDL or RMI-IIOP 40  
     client side 49  
         bootstrap process 51  
         delegation 52  
         getting hold of the remote object 51  
         ORB initialization 50  
         remote method invocation 52  
         servant 52  
         stub creation 49  
     common problems 178  
         client and server running, not naming service 179  
         com.ibm.CORBA.LocateRequestTimeout 178  
         com.ibm.CORBA.RequestTimeout 178  
         hanging 178  
         running the client with client unplugged 180  
         running the client without server 179  
     completion status and minor codes 173  
     component, what it contains 169  
     component, what it does not contain 170

CORBA  
     differences between RMI (JRMP) and RMI-IIOP 45  
     examples 40  
     further reading 40  
     interfaces 40  
     introduction 39  
     Java IDL or RMI-IIOP? 40  
     remote object implementation (or servant) 41  
     RMI-IIOP limitations 40  
     server code 42  
     stubs and ties generation 41  
     summary of differences in client development 45  
     summary of differences in server development 45

debug properties 171  
     com.ibm.CORBA.CommTrace 171  
     com.ibm.CORBA.Debug 171  
     com.ibm.CORBA.Debug.Output 171

debugging 169

diagnostic tools  
     -J-Djavac.dump.stack=1 171  
     -Xtrace 171

exceptions 172

features 55  
     client side interception points 58  
     fragmentation 57  
     interoperable naming service (INS) 60  
     portable interceptors 58  
     portable object adapter 55  
     server side interception points 58

how it works 49

identifying a problem 169  
     fragmentation 170  
     JIT problem 170  
     ORB versions 170  
     packaging 170

ORB (*continued*)  
     identifying a problem (*continued*)  
         platform-dependent problem 170  
         what the ORB component contains 169  
         what the ORB component does not contain 170

properties 46

RMI and RMI-IIOP  
     differences between RMI (JRMP) and RMI-IIOP 45  
     examples 40  
     further reading 40  
     interfaces 40  
     introduction 39  
     remote object implementation (or servant) 41  
     server code 42  
     stub and ties generation 41  
     summary of differences in client development 45  
     summary of differences in server development 45

RMI-IIOP limitations 40

security permissions 173

server side 53  
     processing a request 55  
     servant binding 54  
     servant implementation 54  
     tie generation 54  
     service: collecting data 180  
         data to be collected 181  
         preliminary tests 180  
     stack trace 174  
         description string 174  
         nested exceptions 175  
     system exceptions 172  
         BAD\_OPERATION 172  
         BAD\_PARAM 172  
         COMM\_FAILURE 172  
         DATA\_CONVERSION 172  
         MARSHAL 172  
         NO\_IMPLEMENT 172  
         UNKNOWN 172

traces 175  
     client or server 177  
     comm 176  
     message 175  
     service contexts 177

understanding  
     client side interception points 58  
     features 55  
     fragmentation 57  
     how it works 49  
     interoperable naming service (INS) 60  
     portable interceptors 58  
     portable object adapter 55  
     processing a request 55  
     servant binding 54  
     servant implementation 54  
     server side interception points 58  
     the client side 49  
     the server side 53  
     tie generation 54  
     using 46  
     user exceptions 172  
     versions 170

ORB properties  
     com.ibm.CORBA.AcceptTimeout 46  
     com.ibm.CORBA.AllowUserInterrupt 46  
     com.ibm.CORBA.BootstrapHost 46  
     com.ibm.CORBA.BootstrapPort 46  
     com.ibm.CORBA.BufferSize 46

ORB properties (*continued*)

- com.ibm.CORBA.ConnectionMultiplicity 46
- com.ibm.CORBA.ConnectTimeout 46
- com.ibm.CORBA.enableLocateRequest 47
- com.ibm.CORBA.FragmentSize 47
- com.ibm.CORBA.FragmentTimeout 47
- com.ibm.CORBA.GIOPAddressingDisposition 47
- com.ibm.CORBA.InitialReferencesURL 47
- com.ibm.CORBA.ListenerPort 47
- com.ibm.CORBA.LocalHost 47
- com.ibm.CORBA.LocateRequestTimeout 47
- com.ibm.CORBA.MaxOpenConnections 48
- com.ibm.CORBA.MinOpenConnections 48
- com.ibm.CORBA.NoLocalInterceptors 48
- com.ibm.CORBA.ORBCharEncoding 48
- com.ibm.CORBA.ORBWCharDefault 48
- com.ibm.CORBA.RequestTimeout 48
- com.ibm.CORBA.SendingContextRunTimeSupported 48
- com.ibm.CORBA.SendVersionIdentifier 48
- com.ibm.CORBA.ServerSocketQueueDepth 48
- com.ibm.CORBA.ShortExceptionDetails 48
- com.ibm.tools.rmic.iiop.Debug 48
- com.ibm.tools.rmic.iiop.SkipImports 48
- org.omg.CORBA.ORBId 48
- org.omg.CORBA.ORBLlistenEndpoints 49
- org.omg.CORBA.ORBServerId 49
- org.omg.CORBA.ORBId 48
- org.omg.CORBA.ORBLlistenEndpoints 49
- org.omg.CORBA.ORBServerId 49

OSGi ClassLoading Framework

- shared classes 282

other sources of information xiv

OutOfMemory errors, receiving (AIX) 117

OutOfMemoryErrors, z/OS 162

overview of diagnostics 189

- categorizing problems 189

cross-platform tools 191

- dump formatter 191
- Heapdump 191
- JPDA tools 192
- JVMPi tools 192
- JVMRI 193
- JVMTI 192
- trace formatting 192

platforms 189

## P

packaging, ORB 170

parallel mark (garbage collection) 12

parent-delegation model (class loader) 29

performance

- factors 84
- questions to ask 85

performance problems, debugging

- AIX 119
  - application profiling 125
  - collecting data from a fault condition 125
  - CPU bottlenecks 120
  - finding the bottleneck 120
  - getting AIX technical support 126
  - I/O bottlenecks 125
  - JIT compilation 125
  - JVM heap sizing 125
  - memory bottlenecks 124
- Linux
  - application profiling 138

performance problems, debugging (*continued*)

Linux (*continued*)

- CPU usage 136
- finding the bottleneck 136
- JIT compilation 138
- JVM heap sizing 138
- memory usage 137
- network problems 137

Windows 149

- application profiling 150
- finding the bottleneck 149
- JIT compilation 150
- JVM heap sizing 150
- systems resource usage 149

z/OS 163

- application profiling 164
- badly-performing process 161
- finding the bottleneck 163
- JIT compilation 164
- JVM heap sizing 163
- systems resource usage 163

pers, AIX segment type 105

Pgsp, AIX segment type 105

pid, AIX 102

Pin, AIX segment type 105

platform-dependent problem, ORB 170

platform-specific variations

- core dump 219

platforms supported in diagnostics 189

poor performance, AIX 112

portable interceptors, ORB 58

portable object adapter

- ORB 55

power management 285

ppid, AIX 102

preliminary tests for collecting data, ORB 180

premature expectation 248

pri, AIX 103

printAllStats utility

- shared classes 269

printStats utility

- shared classes 268

private storage usage, z/OS 151

problem determination

- Hewlett-Packard 167
- Sun Solaris 165

problem report

- advice 87
- before you submit 83
- checklist 83
- contents 87
- data to include 83
- escalating problem severity 88
- factors that affect JVM performance 84
- getting files from IBM support 90
- overview 81
  - IBM service 81
  - Java duty manager 81
- performance problem questions 85
- problem severity ratings 87
- submitting data 89
  - sending files to IBM support 89
  - using your own ftp server 91
- submitting to IBM service 81
- test cases 84
- when you will receive your fix 91

problem severity ratings 87

problem severity ratings (*continued*)
   
     escalating 88

problem submission
   
     advice 87

    data 89

        sending files to IBM support 89

        using your own ftp server 91

    escalating problem severity 88

    getting files from IBM support 90

    overview 81

        IBM service 81

        Java duty manager 81

    problem severity ratings 87

    raising a report 87

    sending to IBM service 81

        when you will receive your fix 91

problems, ORB 178
   
     hanging 178

proc file system, Linux 139

process
   
     z/OS

        deadlocked 160

        looping 160

process private, AIX segment type 105

processes section, vmstat command 130

producing Javadumps, Linux 138

properties, ORB 46
   
     com.ibm.CORBA.AcceptTimeout 46

    com.ibm.CORBA.AllowUserInterrupt 46

    com.ibm.CORBA.BootstrapHost 46

    com.ibm.CORBA.BootstrapPort 46

    com.ibm.CORBA.BufferSize 46

    com.ibm.CORBA.ConnectionMultiplicity 46

    com.ibm.CORBA.ConnectTimeout 46

    com.ibm.CORBA.enableLocateRequest 47

    com.ibm.CORBA.FragmentSize 47

    com.ibm.CORBA.FragmentTimeout 47

    com.ibm.CORBA.GIOPAddressingDisposition 47

    com.ibm.CORBA.InitialReferencesURL 47

    com.ibm.CORBA.ListenerPort 47

    com.ibm.CORBA.LocalHost 47

    com.ibm.CORBA.LocateRequestTimeout 47

    com.ibm.CORBA.MaxOpenConnections 48

    com.ibm.CORBA.MinOpenConnections 48

    com.ibm.CORBA.NoLocalInterceptors 48

    com.ibm.CORBA.ORBCharEncoding 48

    com.ibm.CORBA.ORBWCharDefault 48

    com.ibm.CORBA.RequestTimeout 48

    com.ibm.CORBA.SendingContextRunTimeSupported 48

    com.ibm.CORBA.SendVersionIdentifier 48

    com.ibm.CORBA.ServerSocketQueueDepth 48

    com.ibm.CORBA.ShortExceptionDetails 48

    com.ibm.tools.rmic.iiop.Debug 48

    com.ibm.tools.rmic.iiop.SkipImports 48

    org.omg.CORBA.ORBId 48

    org.omg.CORBA.ORBListenEndpoints 49

    org.omg.CORBA.ORBServerId 49

ps command
   
     AIX 102

    Linux 129

ps-ef command, Linux 139

raising a problem report for submission 81

contents 87

escalating problem severity 88

raising a problem report for submission (*continued*)
   
     problem severity ratings 87

RAS interface (JVMRI) 305

RasInfo, JVMRI
   
     request types 315

    structure 314

receive\_exception (receiving reply) 58

receive\_other (receiving reply) 58

receive\_reply (receiving reply) 58

receive\_request (receiving request) 58

receive\_request\_service\_contexts (receiving request) 58

receiving OutOfMemory errors, AIX 117

redeeming stale classes
   
     shared classes 277

reference objects (garbage collection) 15

ReleaseRasInfo, JVMRI 312

reliability, availability, and serviceability interface (JVMRI) 305

Remote Method Invocation
   
     See RMI 75

remote object
   
     ORB client side

        bootstrap process 51

        getting hold of 51

        remote method invocation 52

remote object implementation (or servant) ORB 41

ReportEnv
   
     AIX 97

    Linux 127

    Windows 143

reporting problems in the JVM, summary xiv

request types, JVMRI (RasInfo) 315

RMI 75
   
     debugging applications 77

    Distributed Garbage Collection (DGC) 76

    examples 40

    further reading 40

    implementation 75

    interfaces 40

    introduction 39

    remote object implementation (or servant) 41

    server code 42

        differences between RMI (JRMP) and RMI-IIOP 45

        summary of differences in client development 45

        summary of differences in server development 45

    stubs and ties generation 41

    thread pooling 76

RMI-IIOP
   
     choosing against Java IDL 40

    examples 40

    further reading 40

    interfaces 40

    introduction 39

    limitations 40

    remote object implementation (or servant) 41

    server code 42

        differences between RMI (JRMP) and RMI-IIOP 45

        summary of differences in client development 45

        summary of differences in server development 45

    stubs and ties generation 41

RunDumpRoutine, JVMRI 312

runtime bytecode modification
   
     shared classes 273

runtime diagnostics, class loader 265

## R

raising a problem report for submission 81

contents 87

escalating problem severity 88

# S

Safemode  
    shared classes 274  
sar, AIX 103  
sc, AIX 103  
security permissions for the ORB 173  
selectively disabling the JIT 243  
send\_exception (sending reply) 58  
send\_other (sending reply) 58  
send\_poll (sending request) 58  
send\_reply (sending reply) 58  
send\_request (sending request) 58  
sending data to IBM, Windows 147  
sending files to IBM support  
    outside IBM 89  
    using your own ftp server 91  
sending information to Java Support, Linux 139  
servant, ORB client side 52  
server code, ORB 42  
server side interception points, ORB 58  
    receive\_request (receiving request) 58  
    receive\_request\_service\_contexts (receiving request) 58  
    send\_exception (sending reply) 58  
    send\_other (sending reply) 58  
    send\_reply (sending reply) 58  
server side, ORB 53  
    identifying 177  
    processing a request 55  
    servant binding 54  
    servant implementation 54  
    tie generation 54  
service contexts, ORB 177  
service: collecting data, ORB 180  
    data to be collected 181  
    preliminary tests 180  
SetOutOfMemoryHook, JVMRI 312  
setting up and checking AIX environment 97  
setting up and checking environment, Windows 143, 144  
setting up for dump extraction  
    Windows 144  
settings, default (JVM) 377  
settings, JVM  
    environment 339  
        deprecated JIT options 341  
        diagnostics options 343  
        general options 340  
        Javadump and Heapdump options 341  
severity ratings for problems 87  
    escalating 88  
shared classes  
    benefits 33  
    cache housekeeping 271  
    cache naming 270  
    cache performance 271  
    cache problems 279, 282  
    class GC 273  
    concurrent access 272  
    creating a shared memory area 280  
    creating a shared semaphore 281  
    deploying 270  
    diagnostics 267  
    diagnostics output 267  
    dynamic updates 275  
    finding classes 276  
    growing classpaths 272  
    initialization problems 280  
    introduction 33  
    shared classes (*continued*)  
        jar and zip files 271  
        Java Helper API 277  
        modification contexts 274  
        not filling the cache 272  
        OSGi ClassLoading Framework 282  
        printAllStats utility 269  
        printStats utility 268  
        problem debugging 279  
        redeeming stale classes 277  
        runtime bytecode modification 273  
        Safemode 274  
        SharedClassHelper partitions 274  
        stale classes 277  
        storing classes 276  
        trace 279  
        verbose output 267  
        verboseHelper output 268  
        verboseIO output 267  
        verification problems 281  
        very long classpaths 272  
        writing information in javasharedresources 281  
shared library, AIX segment type 105  
shared memory area  
    shared classes 280  
shared semaphore  
    shared classes 281  
SharedClassHelper partitions  
    shared classes 274  
shmat/mmap, AIX segment type 105  
short-running applications  
    JIT 246  
skeletons, ORB 41  
snap traces 199  
st, AIX 103  
stack trace, interpreting (AIX) 108  
stack trace, ORB 174  
    description string 174  
    nested exceptions 175  
stale classes  
    shared classes 277  
static data 248  
stime, AIX 102  
storage management, Javadump 207  
storage usage, private (z/OS) 151  
storage, z/OS 161  
storing classes  
    shared classes 276  
strace, Linux 139  
string (description), ORB 174  
stusb and ties generation, ORB 41  
submitting a bug report, AIX 119  
submitting data  
    sending files to IBM support 89  
submitting data with a problem report 89  
    sending files to IBM support 89  
    using your own ftp server 91  
summary of differences in client development 45  
summary of differences in server development 45  
Sun properties, deprecated 49  
Sun Solaris  
    problem determination 165  
svmon, AIX 104  
sweep phase (garbage collection) 8  
    detailed description 14  
    parallel bitwise sweep 14

synchronization  
   JNI 71  
 system dump 217  
   defaults 217  
   environment variables 218  
   overview 217  
   platform-specific variations 219  
     Windows 145  
 system dumps 197  
 system exceptions, ORB 172  
   BAD\_OPERATION 172  
   BAD\_PARAM 172  
   COMM\_FAILURE 172  
   DATA\_CONVERSION 172  
   MARSHAL 172  
   NO\_IMPLEMENT 172  
   UNKNOWN 172  
 system logs 129  
 system logs, using (Linux) 138  
 system properties  
   command-line options 363  
 system properties, Javadump 206  
 System.gc() 249, 256  
 systems resource usage, Windows 149  
 systems resource usage, z/OS 163

## T

tags, Javadump 205  
 tat, AIX 103  
 TDUMPs  
   z/OS 155  
 technical support for AIX 126  
 tenure age 19  
 terminology and conventions in this book xiv  
 test cases 84  
 TGC tracing  
   garbage collection 260  
 thread pooling  
   RMI 76  
 threading libraries 128  
 threads as processes, Linux 140  
 tid, AIX 103  
 tilt ratio 19  
 time, AIX 102  
 tool option for dumps 198  
 tools  
   cross-platform 191  
 tools, native  
   Windows 144  
 tools, ReportEnv  
   AIX 97  
   Linux 127  
   Windows 143  
 top command, Linux 139  
 topas, AIX 105  
 tprof, AIX 105  
 trace  
   .dat files 299  
   AIX 106  
   application trace 300  
   applications 283  
   changing options 307  
   controlling 286  
   default 284  
   default Memory Management tracing 284  
   external, calling 316

trace (*continued*)  
   formatter 298  
   invoking 298  
   intercepting trace data 315  
   internal 284  
   Java applications and the JVM 283  
   methods 283  
   MiscellaneousTrace control options 288  
   options  
     applids 288  
     buffers 288, 289  
     count 290  
     detailed descriptions 289  
     exception 290  
     exception.output 288, 295  
     external 290  
     iprint 290  
     maximal 287, 290  
     methods 288, 293  
     minimal 287, 290  
     output 288, 294  
     print 290  
     properties 288, 289  
     resume 288, 296  
     resumecount 288, 296  
     specifying 286  
     state.output 288  
     summary 287  
     suspend 288, 296  
     suspendcount 288, 296  
     trigger 288, 297  
   options that control tracepoint selection 287  
   options that indirectly affect tracepoint selection 288  
   options that specify output files 288  
   placing data into a file 285  
     external tracing 286  
     trace combinations 286  
     tracing to stderr 286  
   placing data into in-storage buffers 285  
     snapping buffers 285  
   power management effect on timers 285  
   properties file 299  
   registering a trace listener 306  
   shared classes 279  
   tracepoint ID 299  
   triggering and suspend or resume 288  
 TraceDeregister, JVMRI 312  
 tracepoint specification 291  
 TraceRegister, JVMRI 313  
 TraceResume, JVMRI 313  
 TraceResumeThis, JVMRI 313  
 traces, ORB 175  
   client or server 177  
   comm 176  
   message 175  
   service contexts 177  
 TraceSet, JVMRI 313  
 TraceSnap, JVMRI 314  
 TraceSuspend, JVMRI 314  
 TraceSuspendThis, JVMRI 314  
 tracing  
   Linux 131  
     ltrace tool 131  
     mtrace tool 131  
     strace tool 131  
 tracing leaks, Windows 148

transaction dumps  
  z/OS 155  
triggering dumps 197  
truss, AIX 106  
tty, AIX 102  
Type, AIX 104  
  clnt 105  
  Description parameter 105  
  mmap 105  
  pers 105  
  work 105

## U

uid, AIX 102  
uname -a command, Linux 139  
understanding memory usage, AIX 112  
understanding the class loader 29  
UNKNOWN 172  
user dumps  
  generating in hang condition 144  
user exceptions, ORB 172  
user, AIX 103  
using dump agents 195  
utilities  
  NLS fonts 184  
  \*.nix platforms 184  
  Windows systems 184

## V

verbose output  
  shared classes 267  
verboseHelper output  
  shared classes 268  
verboseIO output  
  shared classes 267  
verification problems  
  shared classes 281  
versions, ORB 170  
virtual storage, z/OS 161  
vmstat command, Linux 139  
  processes section 130  
vmstat, AIX 106  
Vsid, AIX 104

## W

when you will receive your fix, problem report 91  
who should read this book xiii  
Windows  
  analyzing deadlocks 147  
  classifying leaks 148  
  collecting data 150  
  collecting data from a fault condition 150  
  deadlocks 147  
  debugging performance problems  
    application profiling 150  
    finding the bottleneck 149  
    JIT compilation 150  
    JVM heap sizing 150  
    systems resource usage 149  
  debugging techniques 145  
  diagnosing crashes 146  
    sending data to IBM 147  
  Dump Formatter 145

Windows (*continued*)  
  font utilities 184  
  generating a user dump file in a hang condition 144  
  getting a dump from a hung JVM 147  
  hangs 147  
    analyzing deadlocks 147  
    getting a dump 147  
  Heapdumps 145  
  Javadumps 145  
  JVM dump initiation 220  
  memory leaks 148  
    classifying leaks 148  
    memory model 148  
    tracing leaks 148  
  memory model 148  
  native tools 144  
  performance problems 149  
  problem determination 143  
  sending data to IBM 147  
  setting up and checking environment 143, 144  
  setting up for data collection 144  
    dump extraction 144  
    native Windows tools 144  
  setting up for dump extraction 144  
  system dump 145  
    tracing leaks 148  
  work, AIX segment type 105

## Z

z/OS  
  collecting data 164  
  crash diagnosis 153  
  crashes  
    documents to gather 153  
    failing function 154  
  dbx 153  
  debugging performance problems  
    application profiling 164  
    finding the bottleneck 163  
    JIT compilation 164  
    JVM heap sizing 163  
    systems resource usage 163  
  environment variables 151, 343  
  environment, checking 151  
  error message IDs 153  
  general debugging techniques 152  
  hangs 160  
    bad performance 161  
  Heapdumps, starting 152  
  IPCS commands 152  
  IPCS commands and sample output 156  
  Javadumps, starting 152  
  JVM dump initiation 219  
  LE settings 151  
  maintenance 151  
  memory leaks 161  
    LE HEAP 161  
    OutOfMemoryErrors 162  
    virtual storage 161  
  performance problems 163  
  private storage usage 151  
  process  
    deadlocked 160  
    looping 160  
  setting up dumps 152  
  TDUMPs 155



**IBM**

SC34-6650-01

